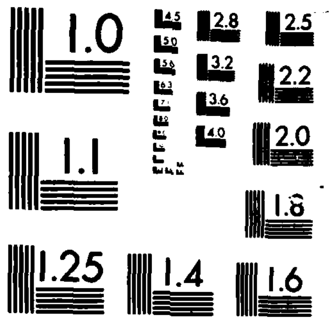
AD-A163 180 FORMALIZATION OF THE INTELLIGENT PROGRAM EDITOR(U) 1/2
ADVANCED DECISION SYSTEMS MOUNTAIN VIEW CA
S G ROSENBAUM ET AL. DEC 85 TR-1047-02 N00014-83-C-0444

UNCLASSIFIED F/G 9/2 NL


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

(12)

ADVANCED DECISION SYSTEMS

FORMERLY:
ADVANCED INFORMATION &
DECISION SYSTEMS

201 San Antonio Circle, Suite 286
Mountain View, CA 94040
FAX (415) 949-4029
(415) 941-3912

TR-1047-02

FORMALIZATION OF THE INTELLIGENT PROGRAM EDITOR

Prepared by:

Susan G. Rosenbaum
William M. Bricken
Michael A. Brzustowicz
Jeffrey S. Dean

December 1985

Annual Technical Report for July 1984 - July 1985

Contract No. N00014-83-C-0444

Prepared for:

Office of Naval Research
Department of the Navy
800 North Quincy Street
Arlington, Virginia 22217

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

AD-A163 180

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) TM-1047-02	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Advanced Decision Systems / 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State and ZIP Code) 201 San Antonio Circle, Suite 286 Mountain View, CA 94040-1289	7b. ADDRESS (City, State and ZIP Code) Department of the Navy 800 N. Quincy Street Arlington, Virginia 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION / 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-83-C-0444
8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO / PROJECT NO. / TASK NO. / WORK UNIT NO
11. TITLE (Include Security Classification) Formalization of the Intelligent Program Editor	

12. PERSONAL AUTHOR(S) Rosenbaum, Susan G.; Bricken, William M.; Brzustowicz, Michael A.; Dean, Jeffrey S.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Annual	FROM 840715 TO 850715	December 1985	104

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB. GR.
09	02	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Programming Environments, Software Engineering, Program Comprehension, Program Knowledge Representation, Intelligent Program Editor, Program Reference Language,

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report covers the work done during the second year of the Intelligent Program Editor (IPE) project. The research this past year focused on four major areas. Two of these areas were concerned with the formalization of two aspects of the IPE: the first was the study of the internal representation needs of the Extended Program Model (EPM) database and the development of a formal structure based on predicate logic; the second was on the formalization of the Program Reference Language (PRL) Picture Language to provide a direct mapping between the Picture Language and predicate logic. Another major research area was in the study of efficient retrieval of query requests from the EPM. The final area of work was in the development of an automatic parsing of Ada programs with subsequent textual and syntactic linking and display on the Symbolics LISP machine. [illegible]

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT [✓] DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr. Robert B. Grafton	22b. TELEPHONE NUMBER (Include Area Code) (202) 696-4302 / 22c. OFFICE SYMBOL Code 430

18. SUBJECT TERMS

Extended Program Model, User Interface, Artificial Intelligence

Table of Contents

List of Figures

1. Introduction

1.1 Project Background

This annual report discusses progress during the second year of a three-year effort of research on an Intelligent Program Editor. Prior to the beginning of this contract, an initial six-month phase studied the development of a prototype database system for manipulating programs [Shapiro 83]. The database and its related functions are known as the EPM, the Extended Program Model. The first year's research of the current contract included the study of the means of interacting with the EPM through the Program Reference Language (PRL) using a pictorial query mechanism [Domeshek 84]. An interface prototyper was developed to provide the ability to simulate the user interface to the IPE. Additionally, editor development was studied in a conceptual framework entitled the Virtual Editing Machine [Brzustowicz 84].

1.2 Summary of Results for Year Two

The work this past year focused on four major areas:

- The internal representation needs of the EPM
- The formalization of the PRL Picture Language
- The automatic parsing of an Ada^{TM} program with subsequent textual and syntactic linking and display
- Efficient retrieval of query requests

After studying the structure of the original EPM, several weaknesses became apparent that would hamper its usefulness to the IPE. To alleviate these problems, an attempt to formalize the EPM was undertaken. A formal structure based on predicate logic has been developed which will ensure the correctness of results obtained from searching the structure. The initial version of this structure has been implemented on a Symbolics 3600 LISP machine. Data and control flow information are explicitly available through the use of this representation.

Related research at ADS on the Program Reference Language has involved work on the formalization of the PRL Picture Language [Bricken 85]. This effort was directed toward the generation of simple, unambiguous queries. A formal version of the Picture Language was developed to address this need. The formal version provides a direct mapping between the Picture Language and predicate logic. Future work will involve the reconciliation of the informal and formal versions of the Picture Language.

Automatic parsing of Ada programs, with the joining of the textual and syntactic representations, was accomplished using programs developed on both the LISP machine and the VAX. Using the LISP machine, a program takes the syntax output from the Surface Parser (SP) (see Chapter 3) developed on the VAX to construct a "tokenizer" to break the program into its token components. After building object representations for the textual and syntactic pieces, the representations are linked internally and displayed graphically as a syntax tree. The user can, with different mouse buttons on the LISP machine, cause the highlighting of the text associated with each syntax node. Likewise, from the text window, the user can highlight a region of text and have the syntax nodes displayed with that region highlighted.

An initial study of search/retrieval was undertaken to study the effectiveness of the IPE in real programming environments. The conclusion from the study is that efficient retrieval is possible, despite the multiple representations used by the IPE. For certain classes of search patterns, the IPE may be capable of outperforming search mechanisms found in many current programming environments.

1.3 Objectives for the Coming Year

The key objective of the IPE project is to provide a prototype system with which to study program manipulation. The work encompasses many parts of software engineering, from display issues to internal representation questions. The plan for the project is to provide a prototype system that exhibits representation linking and query retrieval capabilities for a subset of the Ada language.

The remainder of the project will focus on three majors area of work: improving the parsing capability of the system, refining the EPM, and providing user interface support. A milestone chart is provided in Section 1.4. The following subsections provide brief descriptions on the plans for the final year's effort.

1.3.1 Parsing Enhancements

The first goal of the parsing enhancements subproject is to develop a network between the VAX and the LISP machine to enable us to directly run the Surface Parser from the LISP machine. Currently, the output of the SP is manually ported to the Lisp machine; the network will provide an automatic capability of running the SP on the VAX directly from the Lisp machine and receiving the output immediately from the program. The development of the network will provide a generic tool for VAX/Lisp machine communication.

Another major improvement will be the ability to incrementally parse an Ada program. This ability will be the first step needed towards the capability of updating the EPM after a program modification without a complete reparsing and reformulation.

Additionally, a syntax tree editor will be developed to allow the user to modify the program at either the textual or syntactical level. The text editor used in the IPE is the EMACS [Stallman 81] editor provided on the Lisp machine.

1.3.2 EPM Modifications

The initial task relating to the EPM will be the selection of the subset of Ada that will be recognized by the EPM. It is planned that the Pascal subset of Ada will be the chosen portion of the language. We will also limit EPM representations to single procedures of Ada. We have chosen to address a subset of Ada comprehensively, rather than a full set of Ada shallowly.

The linking of the syntax to the formal EPM will be a major effort. The process will take several steps: the first step will be to convert *loop* statements to their recursive forms. This will serve as the initial testing of the linking procedure. Second, assignments and conditional statements will be combined with converted iterative structures to form a pure recursive structure. From this, we can construct the data and control flow models of the EPM.

1.3.3 User Interface Support

The user interface work on the remainder of the project will be concentrated on two different areas. The first will be the development of a simple query language with which to examine the EPM. The second will be the implementation of LISP machine screen displays for all the EPM representations. It is assumed at this stage of the project that the interface with the system will be via function calls to LISP rather than through a sophisticated interface mechanism.

1.4 Milestones for July 1985 - July 1986

Milestones

TASK	DATE
Select subset of Ada	Oct. 15, '85
Network to VAX	Oct. 31 '85
Syntax linked to formal EPM	
Initial pass, *for loop*	Jan. 15, '86
Assignment statement	April 15, '86
Conditional statement	April 15, '86
Iterative structure	July 15, '86
Pure recursive structure	July 15, '86
Incremental parser	Jan. 15, '86
Design query language	April 15, '86
Syntax tree editor	April 15, '86
Implement query capability	July 15 '86
Display of all representations	July 15, '86

1.5 Report Overview

The next seven chapters of the report provide details as to the accomplishments achieved during the past year. In Chapter 2, a discussion of the overall architecture is given. Chapter 3 provides details as to the development of the Surface Parser Constructor and the syntax and text linkages. A descriptive and pictorial walk-through of a user's interaction with the syntax and text linkages is described in Appendix C.

The original and current EPM models are described in Chapters 4 and 5. Chapter 6 discusses the formalization of the PRL Picture Language. The Mock Engine is related to the picture language; this is discussed in Chapter 7, with a detailed review of the functions provided in Appendix B.

Finally, the study of efficient searching is described in detail in Chapter 8.

2. Architecture of IPE

This chapter is an architectural overview of the current version of the Intelligent Program Editor (IPE), divided into three parts. In the first part, we present the current structure of the IPE. This is followed by a discussion of the capabilities of the system. Finally, we talk a verbal walk-through through a session with the IPE.

2.1 Structure

The Intelligent Program Editor is intended to be a research vehicle which can be used by someone familiar with both Ada and the IPE. Our efforts are therefore aimed less at making the system user friendly at the surface level than at the deep concepts that will allow a truly friendly system to be built when an appropriate interface is added. We anticipate that the user interface at the end of this contract will be *ad hoc* in that many of the IPE functions will have individual interfaces. The structure of the IPE as a whole is thus reflected very strongly in how the user sees the system. There are two important chunks of the IPE that are clearly defined and have visible user interfaces. These are the Extended Program Model (EPM) data base and the Program Reference Language (PRL) editor.

The EPM data base is currently memory resident. It consists of several explicit, cross-linked representations of the program that is being edited. Each representation is created by a separate program. These programs know about each other and can be fired sequentially and automatically when a file is initially edited. Once the EPM is created in memory, it will be maintained incrementally. True incremental analysis of the edited program, akin to incremental compilation but more complex and less well understood, is beyond the current scope of this project. When large programs are involved, incremental analysis is necessary, as rebuilding the entire data base is very expensive. We need to address the issues of incremental data base maintenance for research purposes without solving the incremental compilation/analysis problem. Our solution begins with regeneration of the data base. This is feasible for the small programs we are intending to handle. We then compare the structure of the new and old data bases in order to generate the *same increment* that a true incremental analysis would generate. These update increments are then used on the old data base *as if* they were from a true incremental analysis.

Just as the EPM addresses the modification tasks of editing, the Program Reference Language (PRL) addresses the location tasks of editing and program study. For more information on the PRL, see Chapter 6.

In addition to these major segments, there are some additional features that will be included in the July, 1986 version of the IPE. The most important of these is the inter-representational mapping feature. The user will be allowed to select arbitrary code regions in any representation with the use of the mouse and have those code regions highlighted in any of the other representation.

2.2 Capabilities

The IPE has both user interface and representation capabilities. The user interface provides a separate window for each of three representations of the program being edited: textual, syntactic, and semantic. As mentioned above, the mapping between these representations can be made visible at the user's request by selecting and highlighting with the mouse. There is also a Common LISP Listener window which is used for behind-the-scenes work whenever it may be required. The system is written in Common LISP, which gives the developers direct access to all IPE functions. For example, to read a program into the IPE, a LISP function is executed. The PRL/PL editor will also be invoked from Common LISP. It provides its own pop-up window on invocation and returns a lisp form which is the PRL/FL version of the query. From there, user can modify the query before it is executed.

Currently, three representation tools are defined for the July, 1986 version of the IPE. These are the syntax analyzer, the semantic analyzer and the consistency maintainer. The syntax analyzer is described in detail in Chapter 3. The semantic analyzer is described in detail in Chapter 5. The consistency maintainer cannot be properly specified before the data formats for both the syntax and semantic analyzers are well understood. The algorithmic outline for the consistency maintainer was presented in section 2.1 of this chapter.

2.3 Scenario

We are planning for the July, 1986 version of the IPE to handle a scenario consisting of entering a short Ada program, locating certain code regions with the IPE's advanced search facilities and modifying some portion of the text and syntax views.

For July, 1986, program entry will probably be limited to text mode. The program can either be entered from the keyboard into the IPE text window or can be read in from a previously created file. This latter capability will be used to run the IPE on several of the smaller programs from the Ada test suite. Once the text representation is available to the IPE, the syntactic analyzer will segment the text and generate the syntax view. The syntactic analyzer is a modified version of an LALR parser as created by an augmented LALR parser generator. An LALR parser is a parser that scans textual input from left-to-right, with one token look-ahead used for verifying the state table actions. See Chapter 3 for more details on the parser. The semantic analyzer uses the parse tree created out of this last step to generate the semantic view. This is a four step, continuous stream process. The Ada parse tree is scanned to generate an Ada-like LISP representation. This is then macro expanded into genuine Common LISP. The Common LISP code is processed to remove side effects and generate pure LISP, which is then transcribed into LOSP. For more details on this process, see Chapter 5.

Once the IPE has a program properly loaded into its data base, the user will be able to submit search queries. The method by which the user queries the system will be developed during this next year of research.

The last task in the demonstration scenario of the IPE is the program modification task. Current plans are initially to allow text modifications and to add the capability for syntax modification as time allows. The initial version of the modification task will use the ZMACS text editor, available on the Symbolics LISP machine. Once the program is modified to the user's satisfaction, she/he will manually invoke the consistency maintainer which will generate the increment by the regenerate and compare method described earlier (section 2.1). If the modifications are still legal Ada, the increment will be automatically sent to the EPM to update the data base and then update all the displays. Later versions of the IPE will invoke the consistency maintainer automatically, will work truly incrementally, will support the users interactively in writing correct Ada code, and will allow them to modify the syntax view directly.

3. Syntax View

The work done this project year on the design and implementation of the syntax view of programs is described in this chapter. Apart from this introduction, there are three sections: (1) a presentation of the representation chosen for the syntax view; (2) a discussion of the implementation; and (3) a description of the linked text and syntax views, including possible future work.

The first section describes the representation chosen for the syntax view. The choice of a representation has far-reaching consequences. As a result, the discussion includes possibilities that will not be realized in the foreseeable future, but must be considered nevertheless. The syntax view must be generated automatically from the original program text. In addition, as changes are made to either the program text or the syntax views, the other one must be updated correspondingly. The eventual capability of maintaining data base consistency when a view is modified is made easier by choosing to represent the text view as a series of segments, each of which is meaningful in the syntax view. The text view is segmented into token strings which correspond directly to the leaf nodes in the syntax tree. Characters not included in any token do not convey meaning and may be changed to any other text that similarly does not convey meaning; the syntax view will not be affected under these conditions. Similarly, every syntax node can be mapped into a set of text segments that represent it. This choice of closely coupled representations facilitates keeping both representations in synchronization with each other, allowing for the future possibility of incremental parsing. It does not preclude any useful editing features on either view.

The next section describes how we have implemented the representation described in the first section. It focuses heavily on the parser-developer, which is the key piece of human written software in the system. The parser-developer writes a surface parser for that language when given an accepting grammar for a language. It was developed to take advantage, as much as possible, of existing software. It makes heavy use of the YACC compiler-compiler, which is supplied with the Unix^{TM} operating system. The surface parser, when invoked from the IPE, creates a data stream from which both a syntax tree and a corresponding segmented text list can be built and linked.

The last section of this chapter discusses how the text and syntax views are actually linked together, what capabilities are supported by the joint structure, and possible future work.

3.1 The Syntax View

One of the views of programs that the IPE supports is the syntax view. Because the IPE also supports the view of program text, the syntax view does not have to include all of the program text. It does, however, have to be closely linked to the text view, both for the user's ability to comprehend the system and for the system to properly maintain consistency between the two representations.

The key issues surrounding the choice of representations are the need to have them closely coupled and to have them usable. Because the user can potentially modify either one or both views, it is desirable, in order to maintain consistency, that the representations be bidirectionally linked. In addition, if the syntax view is not going to redundantly include all the program text, then it is necessary from the standpoint of usability that one can make the mappings visible on demand. This also suggests having explicit bidirectional links.

It is easy to see how syntax can point back to text. As the text is being tokenized as part of the parsing process, it is a relatively minor task to instrument the character read routines to provide the offset for the beginning and end of each token in the input stream. But, unless the text is segmented, it makes no sense to talk about a link from the text to the syntax representation. If the program text is a single string (or stream), then that one sting is linked to *all* syntax nodes, which is not useful. The final representation must have *meaningful* links.

Most editors represent text in one of a few, well-known ways. Representing a program by a single string (a limiting case presented above) is an inconvenient strategy, and is not one of the widely accepted representations. Of the commonly accepted approaches, the one that comes closest to the single string representation is the text plus gap editors. Another approach is to segment the text into lines or characters.

Maintaining a program as a linked list of characters is generally considered to be too resource intensive for practical use. However, lines of text do not mesh naturally with the notion of syntactic structure. A given syntax node can refer to a portion of a line, or to portions of several lines, but will, in practice, rarely point to one or more complete lines. The natural solution is to segment the text in a way meaningful to the syntax view, namely into token and token separators.

This is the text representation that we chose; it fits naturally with syntax representations and, as discussed below, is easy to generate and link with it. Since the leaf nodes of the syntax tree are all tokens and are the *only* token nodes, the links between the representation are always to whole units or collections of whole units. This will aid in consistency maintenance and incremental updates; it is trivial to map a change in one representation into the minimum number of changed units in the other. Since the text consists of both token strings and token separator strings, changes in the text which do *not* affect the parse are also easily identified. Finally, the display of the syntax tree can be focused on the nontextual aspects because it is easy to identify and highlight for the user the text that corresponds to a particular syntax unit.

The syntax view is represented as a tree of syntax nodes; the root node corresponds to the program text stream to be edited and the leaf nodes are the tokens of the edited language, in our case Ada. There is no restriction placed on the number of children any node may have, except as may be defined in the edited language. We guarantee in our representation that all tokens end up as leaf nodes in the tree and all leaf nodes are tokens. Each node has a list of text segments on which it depends. This is generated by propagation up the parse tree. Every leaf node has exactly one text segment, and every nonleaf node has all the text segments of all its descendents. Each text segment has corresponding pointers back to every syntax node which depends on it. In addition, to make the syntax representation more accessible to humans, barren stalks are collapsed. Barren stalks are any non-leaf node that does not have two or more children. To preserve the most specific information, a barren stalk is replaced by its child. We have not empirically found an example where the user would want to see the display of a barren stalk, but we retain the information should the need ever arise.

3.2 The Parser-Developer

3.2.1 Motivation

In order to create these two views from straight program text, we needed a non-standard parser, or rather an augmented parser. As none were readily available, we built a parser in house from a public domain LALR grammar by Herm Fischer of Litton Data Systems (HFISCHER@USC-ECLB) for accepting Ada. We used the YACC/LEX compiler writing system, as supplied with the Berkeley 4.2 distribution of Unix, with program sources for the same. We could not simply run the grammar through the system and use the resulting parser as is, nor could we afford to add the needed enhancements by hand for several reasons. The parser created by such automatic systems are not designed for human comprehension and are difficult to read and understand, let alone modify. Also, the grammar that was available to us was a grammar for an Ada accepter--there were no provisions for outputting any kind of parse. Thus, we set about to automate the construction of a parser for a language from an LALR accepting grammar for that language.

3.2.2 Theory

The parser-developer is the result of our research into automatically constructing parsers from accepting grammars. It allows us to build a parser quickly from a bare bones description of the syntax of a language. Although the IPE project is currently focused on Ada, this technique will not only allow us to keep the IPE up-to-date, but also allow us to explore its use with other languages. With this tool, the IPE could be adapted to another language with little more than an accepting grammar for that language.

The parser-developer is closely related to compilers in that it was constructed with a compiler writing system and its input and output programs can be mapped one-to-one onto each other. It is *not* a compiler, however, in that the mapping does *not* preserve semantics--the output program specifies a more complex procedure than the input program.

The grammar for the Ada accepter is written in a language called YACC, which looks like a cross between Backus-Naur Form and the C programming language. Such a grammar is usually called a "YACC grammar", but we need that term for the grammar which specifies the language YACC, which we will talk about in the next subsection.

So, we have an Ada accepting grammar written in YACC, or as a YACC program. This program was originally intended to be compiled by YACC into a machine language Ada accepter. For our purposes, we need a machine language Ada parser. The parser-constructor reads in the YACC program for an accepter and generates a similar YACC program for a parser. That is, the output of the parser-constructor is a YACC program which generates parse trees for the same language that the input YACC program accepts. The parse tree generators written by the parser-constructor already contain all the enhancements needed by our system so no manual intervention is necessary. The original Ada accepting grammar can now be run through the parser-constructor. The resulting YACC program, when compiled by YACC, is the parser that is used by the IPE.

3.2.3 Construction

We modified the compiler writing system available to us so that the parsers that are ultimately developed with it keep track of their position in the program text stream as they tokenize the stream. The manual which describes how to use YACC includes a YACC grammar written in YACC. We used that grammar as a starting point in building the parser-developer. From the YACC accepter grammar, we created a copying grammar, one that would exactly reproduce the input. This was tested on a number of YACC grammars, including itself, to verify that it was correctly recognizing and recreating the YACC programs. From there, it was a relatively straight forward task to modify the YACC copying grammar to augment the code while copying, so that the new code could take advantage of the extensions to YACC to build parse trees. Note that the parser-developer is a YACC program which operates on an input YACC program and writes a new YACC program which bears a functional relationship with the input program.

3.3 Applying the Parser-Developer

Once the idea of the parser-developer is realized, the details of how the resulting parsers (which are its outputs, given well-formed inputs) behave is an independent issue. Our parser works by following the internal state of the LALR engine created by YACC for a given input. Every explicit shift transaction and explicit reduce transition causes a record of that transition to go to the output stream. Unfortunately for the cleanliness of the implementation, YACC's output programs frequently do implicit shifts. Thus, some of the shift transitions that should be a part of a following reduce will be absent from the output stream. However, since the internal state numbers are available at shift transitions, and a list of all the state numbers that should be popped, and their exact order are available at reduce time, this deficiency is easily resolved.

The output of the parser, as mentioned, is a stream of records of shift and reduce transitions. A LISP function within the running IPE reads these records and builds the actual parse tree data base from them. As it reads the stream, it keeps two stacks. The first one is a stack of syntax nodes which form the tree. The second is a stack of the state numbers that represent the state the YACC-base parser was in for that transition. The two stacks are always run in complete synchronization. Shift transitions push a new item onto each stack. Reduce transitions pop a number of items off each stack and push one new item onto each as well. The syntax nodes popped off the stack are the children of the newly created reduce node. When the process is finished, the top of the stack is the root of the parse tree. The deficiency mentioned in the previous paragraph is resolved at reduce time. The reduce record contains a "pop" list of states expected on the stack in the order they are expected to appear. The tree builder iterates down this list. Whenever the state at the top of the dual stacks matches the current state in the pop list, it is accumulated as a child of the new syntax node being created. Otherwise, the entry in the pop list is ignored. The pruning of null transitions, which the YACC-based parser insert as non-token leaf nodes, and the collapsing of barren stalks, mentioned above, occur at this time as well.

The shift transition record for tokens also includes the position within the program text stream that the token starts and end on, as well as the value of the token. Because of the nature of LALR parsers, these always occur in the same order as they do in the text file. So, while the parse tree is being built, so is a list of text segments to accommodate the tokens. As this list is built, the text segments in it are being bidirectionally linked to the syntax tree. Once the parse tree is finished, the list of text segments, which is already in the right order, is examined and filled in with text segments to hold the values of the token separators. The program text stream is then read in order to fill in the missing strings. Once that is done, the two representations are complete and properly linked, and ready for use.

4. Design of the Original EPM

4.1 Overview

Before studying the new Extended Program Model being developed for the IPE, it will be helpful to examine the original EPM. The original version of the EPM was developed in FranzLisp on a VAX 11/780. The code was written using the *defstruct* mechanism for developing the data structures used to define the program knowledge base. It should be emphasized that the original EPM was developed as a prototype system to handle one small Ada program. Therefore, although the database was an effective vehicle with which to study the problems involved in maintaining multiple pieces of linked information, it was not meant as the final solution needed for the EPM representations.

The first step in studying the design and functionality of the original system was to port the code to the Symbolics LISP machine under Zetalisp. The transition required recoding of the EPM into the new dialect, including some small modifications to the data structures to cause them to work correctly. The EPM was then recoded into the object-oriented *Flavor system* of the LISP machine to take advantage of the ability to build objects that would inherit characteristics from their component objects; this was used in building the multiple data representations with a common base so that all would have an underlying compatibility. Once this work was completed, the actual study of the system began.

4.2 Structure

The desired goal of the representations in the EPM is to provide the ability to study a program textually, syntactically, semantically, and through documentation. The textual representation can be derived from the actual text of the program; the syntactical representation is developed from the syntax tree; semantic information is found in both data and control flow and typical programming patterns, also known as cliches; and documentation is built through user-supplied commenting of code. Four of these program representations are in the original EPM prototype:

1. Text

2. Syntax

3. Combined data and control flow, known as a segmented parse

4. Cliches

To facilitate the translation from one representation to another, a fifth structure known as a code region was used internally in the system to consolidate multiple syntax nodes. This structure was then widely used as the mapping mechanism between the forms. The structure of each of the four major representations is as follows:

- *Textual*-- The text string representation is the usual representation found in most editors, consisting of words and delimiters. The structure for the text representation of the program is a linked list of text line structures. Each structure contains a full line of text. The text node structures contain pointers to the syntax structures. All mappings to other representations must be done by an initial mapping to the syntax structure, with other mappings being derived from the syntax.

- *Syntactic*-- The syntax nodes are used to represent the syntactic parse of the program formed by using the rules of grammar for the programming language. The nodes contain pointers to their predecessors and successors in the parse and pointers to the textual, segmented parse, and cliche representations. The mappings from the syntactic representation to segmented parse, text, and cliches is contained directly within the data structure for the syntax node; no intermediate steps are needed to transform from a syntax node to another form.

- *Segmented Parse*-- The segmented parse structure contains input and output information used for constructing data flow and linkages between structures for control flow. Data flow studies the passage of variables and constants through a program. Control flow studies the flow of program execution. The segmented parse maps to other representations by first converting to the appropriate code regions (multiple syntax nodes) and from there to the other representations.

- *Cliche*-- The cliche structure is a stand-alone structure. Cliches are combinations of objects that perform a known function in a program. Examples of a typical cliche would include a portion of code to do a quicksort or a binary search. The mapping of a cliche into another representation is handled similarly to the mapping procedure used by the segmented parse. The code regions for the cliche are developed; these are then used as the mechanism by which to retrieve the other forms.

4.3 Queries

Queries on the original EPM required the user's deep understanding of the manner in which the system was built and the queries that could be addressed. All of the queries on this version of the EPM were developed by the user typing LISP functions. The system would return a uniquely named object that was a set containing the structure that satisfied the query. The user could then utilize the name in further delving into the system.

Each of the structures that comprises the EPM has a unique name that references the structure itself. For example, a text region would be named *tr1*. Typing simply this symbol would return:

```
TEXT00973:
 loop for MAXSIZE in 1..10
   TOTAL:=ARRAYSUM(TEST_SCORES, MAXSIZE);
   put(TOTAL);
 end loop;
```

Initial queries of the system could be used to examine these pieces. More complicated queries would involve converting one representation into another. A pertinent example would be the query used to convert a cliche into its associated text representation. The query to solve this problem assumes that the user has the name of a cliche, *TPP1*, in this example. The query would be:

```
(tpp-to-crs TPP1)
```

The system would return the text in the program that contained *TPP1*:

```
procedure TEST (A:in INTEGER) is
   begin
    A:=A+5;
   return A;
end TEST;
```

A speed improvement in the searching of the EPM was obtained by forming small databases called the *text-database*, *segp-database*, and *cliche-database* which contained pointers to the structures of each of those types. Structures with particular characteristics could be found by querying the relevant database. For example, the user can search for a particular text token *TEST-* in the text-database with the request:

```
(find-token "TEST-" text-database)
```

The system would return any text string matching this token. For a more complicated query example, the user could ask for any cliches of the type *summation* in the cliche-database with the query

```
(index 'summation cliche-database)
```

The result of the query would be all of the cliches that were of the requested type.

4.4 Searching

The search mechanism used in the original prototype version of the EPM was a simple matching algorithm that looked for particular datatypes or structures. For example, the searching required for any of the *index* or *find-token* queries was merely an access into a hash-table of elements that had been previously entered by a programmer. The search used to find a structure of a particular data type was a two-part process; the system first aggregated all structures that were to be searched into a set and then applied a simple pattern matching test to each of those structures to find the ones that contained the required information. An example of this can be found in the query:

```
(*apply (segs-within SET1) '(seg-type "initialization"))
```

The parameter *SET1* is assumed to be a set of data structures that was previously constructed containing all of the relevant structures for the **apply* query. The search path for finding the cliches of type *initialization* would be to go through each of the elements of the set and attempt to match its type with the type *"initialization"*. Thus, a very simple matching mechanism is being used based on matching the type of a particular structure. The initial EPM was not able to handle the matching of an entire data structure in response to a pattern nor could it operate on a query with several matching sequences to be used simultaneously over the database.

4.5 Problems

The original EPM data structure worked well for a prototype system, but difficulty was found when extensions of the system were attempted. The database on which the system worked had been prepared by hand, a slow and tedious process. While attempting to build tools to automate this process, it was discovered that the data structures were adequate for the small program on which they operated, but inadequate for larger, more complex programs.

A good deal of the difficulty with larger programs was in the need to be able to extend the structures to handle more complex program components. The use of the *Flavor system* as the underlying representation was quite adequate for studying the initial program representation problems, but it was soon discovered that a formal representation would be needed to handle more complex queries and program components. For this reason, the formal EPM was designed. The textual and syntactic forms of the system will still be represented using the *Flavor* data structure; however, the semantic representation will be a mathematically based formal logic structure discussed in the next chapter.

5. The Extended Program Model

The Extended Program Model (EPM) is a set of representations of the source code. It is central to the IPE since it is the stored representation that is the query database. Were the source code stored solely as text, only textual queries would be possible. This is this case for simple editors. The EPM, however, is intended to permit queries not only of textual structure, but also of syntactic and semantic structure in the code. For this reason, the EPM is an extended representation; it models the code at several abstract levels. The EPM currently includes three abstract representations: the textual copy, the syntactic parse tree, and the semantic model. We are able to automate the construction of these representations from the source code. A fourth model, the documentation text model, includes text that is not an integral part of the code itself.

The documentation level contains comments and notations excluded from the functional representation of the code. Documentation can serve multiple purposes:

- to describe the operation of the code.
- *to describe the organizational structure of the code.*
- to describe the connectivity of the code, its interface with other modules.
- to describe the intended uses of the code, its pragmatics.

The long-term design of the EPM includes incorporation of the model of the code that the author has, in the form of structured documentation. This model will permit semantic correction and guidance by the editor while the user is constructing code from his specification/model. The design of the documentation model is not discussed in this report.

We present our technical approach to the definition of the EPM in the next section. Following that is a discussion of the formal models embodied in the EPM. Finally, we present a detailed example of the automated conversion of a sample piece of Ada code into the various parts of the EPM. As yet, we have automated only portions of this process; we expect to have full automation within the next research year.

5.1 The Modeling Goals

Our goals for developing the EPM are:

1. to design a fully automated conversion process that constructs the different models of the EPM from the source code,

2. to maintain a formal rigor in the specification of the component models, and

3. to assure that search, retrieval, and modification of the EPM database is both possible and efficient.

The technical characteristics that we wish to incorporate into the EPM are:

1. All control and reference must be explicit.

2. The complexity of the representation must be minimal, while the functionality of the code must be invariant, and

3. Following the lead of logic programming languages, control and description of the code must be expressed independently of one another.

Our design and formal specifications for the EPM achieve these technical objectives. However, we found it necessary to partition the problem and to approach smaller portions rather than striving for total generality. Therefore, we have adopted two research constraints:

1. We chose to use LISP as the basic research language. This decision is not unprecedented; LISP is the primary research language of the AI community. To illustrate the connectability of the EPM to other languages, we are developing an Ada to LISP conversion program. We chose Ada as our target language due to its prevalence in the military community.

2. We chose to limit our initial implementation to basic programming constructs, specifically: assignment, conditionals, and repetitive application (iteration and recursion).

5.2 The EPM Models

The four models incorporated in the design of the EPM are the Text, the Syntactic, the Semantic, and the Documentation models.

1. The TEXT is a literal model, its representation is a sequence of tokens and token separators.

2. The SYNTACTIC model is a tree representation of the textual source incorporating atomic structures (both names and reserved tokens) as nodes, and syntactic relations as arcs.

   This syntactic parse tree provides structural information about the representation rules of the source language. In our examples, this language is Ada. By creating a syntactic parser for an alternative language, we can map program representations from different languages onto a canonical form. That is, the EPM is generic across languages. To modify it for other languages, all that is needed is the formal grammar of the language, (for the syntactic parser), and a grammar to canonical form mapping, to construct the functionally equivalent expression in the canonical research language. Naturally only subsets of each target language can be accommodated by the EPM. The EPM is not *complete over all languages*.

3. The SEMANTIC model represents the control structure of the textual source, the data structure of the source, and the functional relations between data and control.

   The model of procedural semantics provides both the power and the uniqueness of the PRL paradigm, since it allows the user to express queries in terms of the functionality of the code, rather than solely in terms of the textual structure.

4. The DOCUMENTATION model includes comments (textual documentation) and pragmatic knowledge and facts (structured documentation) about the code.

5.3 Design of the Semantic Model

The semantic model is currently in a design phase. An outline of our design considerations follows.

The construction of the semantic model incorporates three phases:

1. Explicit transcription into the research language (LISP),

2. Minimization of the representational form, and

3. Orthogonalization of data and control.

The conversion process is expected to involve several intermediate steps and several intermediate representations. Although the final conversion process will not need every step, we know that it is important for design to make small incremental changes to the representation, so that we can build and debug modularly.

The sequence of representation languages used in the construction of the semantic form is:

1. Textual Ada: the syntactic and textual models of the source program,

2. Textual LISP: a literal mapping of the syntactic and textual models from Ada to LISP,

3. Elementary LISP: macro conversion of special forms (such as LOOP) into the elementary lexicon of LISP,

4. Pure LISP: iterative forms become recursive forms; assignments become argument bindings within functions,

5. Pure Mathematical LISP: accumulation forms are absorbed into argument forms (this step may be unnecessary),

6. LAMBDA calculus: binding environments are made explicit,

7. LOSP: The LOSP language is a formalism in which all data structures are expressed as descriptive atoms, and all control and Boolean structures are expressed as nestings of parentheses, thus achieving total separation of data from control without loss of either expressive power or functional performance.

For a full description of one version of LOSP, see Spencer-Brown's seminal mathematical text, LAWS OF FORM. We will not attempt a description of this work here.[1]

The construction process requires these explicit translation steps:

1. EXPLICIT TRANSCRIPTION:
   a. Literal Ada to LISP conversion
   b. Make implicit structure explicit
2. MINIMIZATION:
   a. Literal iterative to recursive conversion

[1]The inclusion of LOSP into the code for the EPM is a tentative design decision at this time, since LOSP is copyright-protected software.

b. Remove invariant structures

c. Remove accumulation structures

3. ORTHOGONALIZATION:

a. Literal conversion to LAMBDA form

b. Separate control from data by LOSP conversion.

The following subsection contains an example of the translation process to convert an Ada program into LOSP form.

5.3.1 Transcription of Ada Text into the EPM, An Example

The Ada form at Step 0 is the textual representation in the EPM. The syntactic parse tree that is used to convert Ada syntax to LISP syntax at Step 1 is the syntactic representation in the EPM. The remaining steps describe the construction of the semantic representation which is in the representation language LOSP at Step 7.

STEP 0: The ADA Source Code Example:

```
PROCEDURE TESTADA (X: in integer) is
  pragma MAIN;
  I, B:  integer;
    begin
      B := 0;
      for I in 1 .. X loop
        if I < 5 then
          B := B + I;
        else
          B := B - I;
        end if;
      end loop;
    end TESTADA;
```

STEP 1: Literal Translation into LISP:

```
(defun LADA1 (X)
  (and (integer X)
       (setq B 0)
       (loop for I from 1 to X
         if (< I 5)
           do (setq B (+ B I))
         else
           do (setq B (- B I))
         finally (return B)
         ) ) )
```

STEP 2: Explicit LISP Form:

```
(defun LADA2 (X)
  (and (integer X)
       (do ((I 1 (add1 I))
            (B 0) )
           ((or (< X 1) (> I X)) B)
           (if (< I 5)
               (setq B (+ B I))
               (setq B (- B I))
               ) ) ) )
```

STEP 3: Literal Recursive Representation:

```
(defun LADA3 (X)
  (and (integer X)
       (LADA3-AUX X 1 0)
       ) )

(defun LADA3-AUX (X I B)
  (cond ((or (< X 1) (> I X)) B)
        ((< I 5) (LADA3-AUX X (add1 I) (+ B I)))
        (t (LADA3-AUX X (add1 I) (- B I)))
        ) )
```

STEP 4: Remove Iteration Variables:

```
(defun LADA4 (X)
  (and (integer X)
       (LADA4-AUX X 0)
       ) )

(defun LADA4-AUX (X B)
  (cond ((< X 1) 0)
        ((< X 5) (LADA4-AUX (sub1 X) (+ B X)))
        (t (LADA4-AUX (sub1 X) (- B X)))
        ) )
```

STEP 5: Remove Accumulation Variables:

```
(defun LADA5 (X)
  (and (integer X)
       (cond ((< X 1) 0)
             ((< X 5) (+ (LADA5 (sub1 X)) X))
             (t (- (LADA5 (sub1 X)) X))
             ) ) )
```

STEP 6: Lambda Form:

```
(bind LADA6
  (lambda (X)
    (and (integer X)
         (cond ((< X 1) 0)
               ((< X 5) (+ (LADA5 (sub1 X)) X))
               (t (- (LADA5 (sub1 X)) X))
               ) ) ) )
```

STEP 7: LOSP Removes Control functions:

```
(bond LADA7
  (lombda (X)
    ( ((integer X))
      (((< X 1)) 0)
      ( (< X 1)   ((((< X 5)) (+ (LADA7 (sub1 X)) X))
                   ( (< X 5)  (- (LADA7 (sub1 X)) X))
                  )) ) ) )
```

6. The Formal Picture Language

A major thrust of the IPE project is to develop a query language and an accompanying interface that simplifies the submission of program queries to the EPM. For simple requests, the pictorial language described in [Domeshek 84] permitted the user to access the EPM database by creating two-dimensional representations of logical structures that contained syntactic identifiers.

The simple picture language suffered from a serious defect: it was inconsistent. This defect is intolerable for these reasons:

1. Complex requests were ambiguous, and could not be translated into consistent queries. Representations did not map one-to-one onto specifications.

2. The user would be reinforced to think about program specifications that could not be expressed in a formal language. The user's model of the system would be undermined by inconsistency.

3. The language was incomplete, forcing additional relations. For example, both CONTAINS and IS-CONTAINED-BY were necessary.

4. Some operators, such as negation, produced counter-intuitive results. Object tokens became confused with logical values.

Thus, our research direction focused on the formalization of the Picture Language.

6.1 Formal Specification of the Picture Language

There are two simple, formal, pictorial representations of logical structure, which are dual to each other. Table 6-1 specifies our formal pictorial logic. Parentheses should be read as geometrically surrounding figures, such as circles.

a	==>	a
a AND b	==>	a b
NOT a	==>	(a)
a OR b	==>	((a) (b))
IF a THEN b	==>	(a (b))

Figure 6-1: The Formal Picture Language

CONJUNCTION is achieved by placing tokens within the same representational space. No token of conjunction is used. That is, conjunction is implicit in the space of reference.

NEGATION is expressed by the encapsulation of a variable, surrounding it by a geometric figure, such as a circle, or in a linear representation, by parentheses.

OR and IF can be seen as logical/pictorial compositions of their definitions in terms of AND and NOT. Specifically:

```
a OR b  =  NOT ( (NOT a) AND (NOT b) )
        =      ( (     a)     (     b) )   =   ((a)(b))
```

Since AND is implicit in the PL representation, and since NOT is redundant with the parentheses, the above expression simplifies to a form that contains no explicit conditional or control functions.

Similarly, IF is constructed as follows:

```
IF a THEN b  =  NOT ( a AND (NOT b))
             =      ( a     (    b))      =   ( a (b))
```

This very simple transformation from the representation of logical connectives as tokens to their representation as figures has two remarkable properties:

1. The representation requires only one (super-dimensional) token, that of the parentheses or the circle.

2. The representation is isomorphic with Propositional Calculus.

In this representation, the logical forms of an expression are stored in a higher dimensional space than are the variable forms. The consequence is a single operator logic than is not representationally explosive (as is, for instance, the Sheffer stroke). Since geometric representations do not interact with linear representations, the expression of control as parentheses is independent of the expression of data as variables.

6.2 Representation of Containment

The primary relation that the Picture Language is intended to embody is that of CONTAINMENT. To achieve this pictorially, we overloaded the representation of IF to also mean CONTAINS. Thus

```
( a (b) )
```

is the representation for both "IF a THEN b" and "a CONTAINS b". The intuitive mapping is analogous to the mapping between "b IS-A-SUBSET-OF a" and the Venn Diagram expressing this relation, abstractly: (a (b)).

To alleviate the necessity for inverse relational expressions (CONTAINS and IS-CONTAINED-BY), we highlight the expression intended as the focus of search. Thus:

```
( a ( *b* ))
```

specifies "Find the Bs contained-by A." Conversely,

```
( *a* ( b ))
```

specifies "Find the As containing B."

To summarize our representation techniques for the Picture Language:

1. Boolean operators are expressed by two dimensional figures surrounding tokens.

2. CONJUNCTION is implicit when forms share the same representative space.

3. NEGATION is represented by containment in a geometrical figure.

4. Other logical connectives are defined in terms of AND and NOT.

5. The CONTAINMENT relation is overloaded on the representation of IF.

6. The focus of search is highlighted.

6.3 Examples

Appendix A re-examines 18 query requests in the informal picture language from the perspective of the formal picture language. This Appendix includes:

- A formal specification of each query expressed in Relational Logic,
- A search program that achieves the request, written in an abstract version of LISP,
- The evolutionary development of the EPM search form of the query. The reader can trace a specification of the algorithm that translates an EPM query into a query expressed in Relational Logic,
- An abstract representation of the output of the search procedure.

Appendix A also contains working comments and notes that were developed during this translation exercise.

6.4 The User-PRL Interface

The intent of the PRL is to provide the user with a friendly interface with which to make requests about program structure. The interface is currently being fully redesigned.

6.4.1 Design Considerations

Our design focus has been to identify those conventions that both maintain a mapping onto the formal language and encourage an ease of expression of the informal intentions of the user. Several ideas need to be evaluated:

1. MIXED REPRESENTATIONS: Users should be able to freely mix pictorial elements of the PRL with string identifiers from written English. Further, users should be able to construct requests using mixed representations: part pictorial, part informal English, part formal logic. The request parser will convert these to expressions in a single formal language, either logic, code, or pictorial.

2. AUTOMATIC SIMPLIFICATION: Since the EPM form is a canonical representation of the source code, the system will be able to identify awkward constructs and redundant logical structure. The user could be notified of:

   clumsy control structure and how to improve it,
   clumsy and redundant variables and how to minimize memory,
   clumsy and confusing coding styles and how to clarify them.

3. INSTRUCTABILITY: For the request parser to understand informal (and even inconsistent) requests, it will need to be able to enter into a dialogue with the user about the intent of the request. The parser will default and/or coerce requests into a formal specification, notify the user of what has happened, and request conformation or clarification of the request. Additional techniques include maximal display of information and continual feedback.

4. FRIENDLINESS: We are developing a coordinated package of user-friendly display and interface options that inform the user of the interpretation that the PRL places on a request. These include:

   a. *Multiple parsing languages*: The user can see the request expressed in a variety of representations (for example: English, EPM, logic, search code).

   b. *Multiple access techniques*: Request items can be typed, mouse-clicked, menu-selected, or defaulted.

   c. *Dynamic parsing display*: The request is displayed in terms of both a formal language and the expected results. Requests are dynamically redisplayed when altered, and dynamically simplified when possible.

   d. *Empathetic searching*: Using AI techniques, the interface can refer to a model of the intentions of the user, displaying queries and notifications whenever the request does not meet the modeled intents. For example, if a request returns 400 instances, the interface would ask the user if that was intended, or should the request be further constrained.

   e. *Dynamic display of dynamic processes*: The user has the option to see the search process unfold dynamically instead of the traditional static display of request then results.

6.4.2 Notes on the Design of the Display for the Picture Language

Our design considerations and the following display draw from our experience with Mock-Engine as a prototyping tool.

6.4.2.1 Rough Screen Design

A rough design of the editing screen is included as Figure 6-2. This design is not to scale. It is intended to illustrate the components needed at the visual interface level.

The WORK AREA of this screen is composed of panes that display the following:

- Selection menus for
  - quantification
  - parsing language
  - logical operators
  - relations
  - object types
  - focusers, highlight, ...
  - top level control
- The current construction environment,
- The current parsing of construction,
- The returnables under the current construction,
- Pop-up windows for rarely used items.

6.5 Additional Representation Issues

We conclude this chapter with a brief discussion of several issues of screen representation and functionality.

1. PROPER NAMES: The user should be able to specify a proper name, and the system furnishes the type. Note that only VARIABLES and FUNCTIONS have proper names. Display of proper names looks the same as display by clicking the name slot, i.e.:

   function
   name: foo

2. CONSISTENCY OF VARIABLE LABELS: If a label is used more than once in the same configuration, it refers to the same object. If the user wishes to specify two different functions, for example, he should say FN1 and FN2.

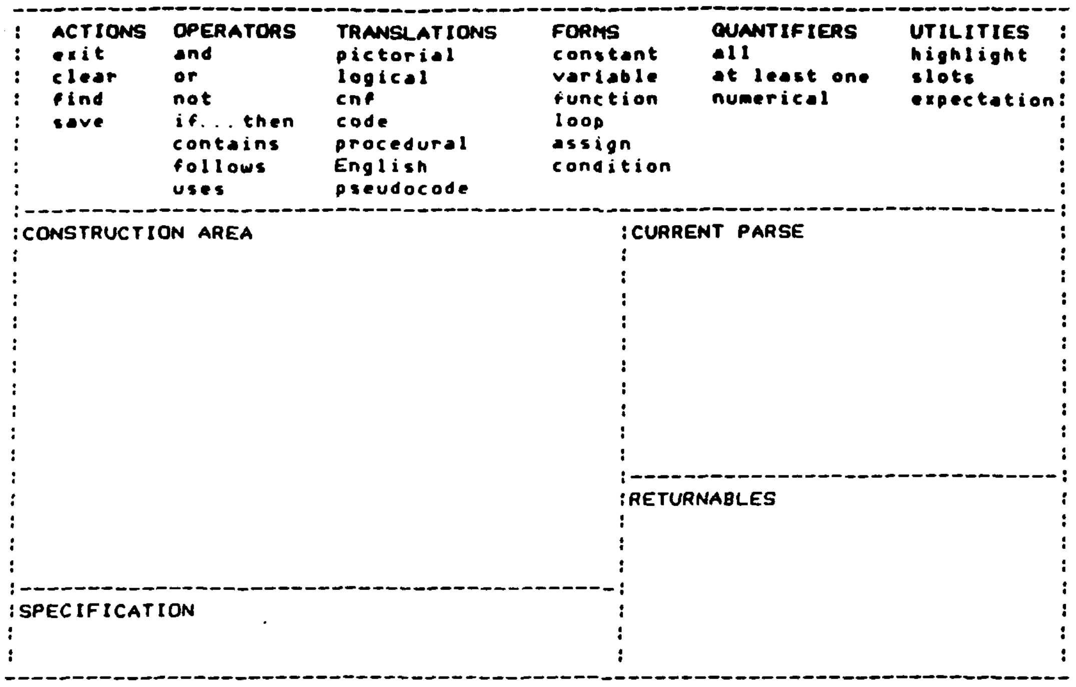


Figure 6-2: Rough Screen Design for the Visual Interface

3. OBJECT/ATTRIBUTE STRUCTURE: Each type of object has an attribute structure. It may be possible to include a clickable menu of common object types, but in any event, clicking on an object type label should pop up a menu of available attributes. Clicking on a particular attribute label should add the attribute name under the object variable, and position the cursor ready to enter a particular specification for the attribute. For example (@ is "click"):

   STEP 1.

   @ function

   STEP 2.

   function
   name: <cursor here>

   STEP 3.

   function
   name: foo <typed by user>

   The value of the attribute field called *name* is now FOO.

4. *QUANTIFICATION: When an object label is entered, and the user types a* CARRIAGE RETURN, a quantification field label, "#:" appears under the object label, followed by the cursor. Options are ALL, a number, THE, or COUNT. THE means the single one. COUNT means to determine how many instances are in the database by counting. Invalid specifications would give an error message, and allow another attempt.

   Non-quantified variables will default to universal quantification. The highlighted object, the focus of the search, is also assumed to be quantified universally. For example:

   Find the functions that contain a loop.

   becomes

   `(loop over all functions, find ALL loops).`

   The system should report the number of instantiations, especially if it gets large (say >20), in a warning message. The system should ask:

   There are at least 45 instances of this request. Do you want them all, or do you want to further constrain the search?

5. NEGATION: The negation of the relation CONTAINS, *DOES NOT CONTAIN* requires the existence of the container and the universal non-existence of the contained. The prototype quantification is:

```
All X Not Exists Y . (contains X Y)
```

which says that All X contain no Y. The Y does not exist. The quantifier negation can be transferred to the relation by transformation rules, resulting in

```
All X All Y . not (contains X Y)
```

which says, equivalently, that All X do not contain any Y.

This should be the default whenever the user specifies not-contains.

The representation of NOT-CONTAINS in the PL cannot cancel the containment:

```
[[ fn [lp] ]]
```

is needed, the double bracket being NOT-CONTAINS.

6. TRANSFORMATION RULES: EPM transformation rules specify alternative display forms. The primary rules are:

   DISTRIBUTION: a [[b] [c]] <==> [[a b] [a c]]

   FLEX: [[a [b]] [[a] c]] <==> [a b] [[a] [c]]

   As well, some "automatic" transformations convert complex expressions into equivalent simple ones:

   IDEMPOTENCY: a a <==> a

   REFLECTION: [[a]] <==> a

   PERVASIVENESS: [a b] b <==> [a] b

7. MIXED VOCABULARY: These languages are available to express the user's needs:

   - Common English: Find the functions that contain loops.
   - Formal Logic: All FN Exists LP . (contains FN LP)
   - EPM: [FNx [LPx [fn [lp]]]]
   - Pseudo-code: (find FN (contains FN LP))

   Users should be able to mix these languages to a moderate degree.

   - Mixed: if [fn [lp]] then fn

   The display work area can be sensitive to objects that are labels, regardless of language, and try to parse them into a comprehensible whole.

8. CLICKABILITY: generally all construction should be reachable from menu selections. The exception would be proper names.

   All clicks should be SMART. When a user clicks a selection, the system should construct as much of the desired form as possible. A template would be displayed with as much information as is supported by the intelligent processor filled in. The user would complete the form. An example sequence might be:

   a. user clicks OR,

   b. [[ ] [ ]] is constructed in work area (recall the PL parsing regime),

   c. cursor goes to *: in [[*] [ ]] and waits for label entry,

   d. after CR, cursor prompts for additionals such as quantification,

   e. slots, etc., after final CR, cursor goes to next item, eg: [[foo] [*]].

9. EXPECTATIONS: The system should display its expectations whenever a request is entered. This is a form of verification.

```
Request:          ''Find the functions that contain loops.''

System:           ''The expected form of the results is:

                        (  (FN1 (LP1 LP2))
                           (FN2 (LP3))
                           ... )

                        ''OK?''
```

7. Interface Prototyper

7.1 Overview

Work on the Intelligent Program Editor has illuminated the need for efficient presentation of information to the user. One of the problems that has been observed is that the user is often unsure as to the best way to initially interact with a new system. The IPE is built upon a large, complex database that contains the internal representation of a program in a variety of forms, including textual, syntactical, and data flow forms. A user's transversal of the database is quite time-consuming; currently there exists only a crude LISP-based interface with which to painfully work on the design of a system query. Often many different requests are needed before the user is able to find exactly the code for which he has been searching.

The study of this problem resulted in the idea of a request system that used pictures rather than words to describe a database request. While working on designing the actual graphic shapes to be used in forming a request, it was decided that a system was needed with which one could visually see what forms the requests would be taking. This requirement evolved into an interface prototyping system, called the Mock Engine that could be used for quickly developing the visual presentation that would be supplied to the user. (For more information on the commands of the Mock Engine, see the appendix.) Not only could this facility be used for the immediate need of the IPE work, but it could also be used for quickly developing many other user interface representations long before coding began; this would allow the designer to easily modify his ideas based on others' inputs without the problem of having to change the actual system based on their responses.

In addition to being able to study the display presentation, the system was also designed so that the process of a user sequencing through a set of commands could be emulated. The system developer can create a group of displays that would illustrate a set of user interactions. The displays can then be saved on a file to be read into the system and viewed in a movie-like fashion.

A supplementary capability is provided by the screen dump facility which allows the designer to print an image of a monochrome screen onto the laser printer. This allows design modification and critiquing to occur both on- and off-line, as well as providing for the production of slides from the black and white display.

7.2 Design

The interface prototyper consists of two basic components: a simple graphics editor with which to develop screen pictures and a mechanism for storing either one image or a sequence of images for later display. The graphics editor allows the user to place shapes on a screen and then manipulate these shapes through a series of pre-defined commands.

The shapes provided by the graphics editor include circles, rectangles, and arrows. Each of these objects can have his position changed and size modified. The rectangle handles more complex commands. It is actually a small editor window that allows the user to add text using the normal EMACS command sequences. Two other modifications the user can make to the rectangle shape are to alter its borders and background shade. The user can increase or decrease the size of the borders of a rectangle and have a gray-scale pattern as its background rather than the default white background.

The shapes are created and modified through menu selections from a command menu located along the right-hand side of the screen. The user would first select a shape to be generated, place the shape in the desired location, and then select from the existing commands to modify the shape.

Once the user is satisfied with a display, he can ask the system to save it as an individual unit on a file, or he can insert the display into a list of displays that the system keeps internally. The saved displays can be saved onto a file; the system will store enough information about the individual graph layouts to enable their re-creation. The list of displays provide the interface prototyper's movie-like "animation" capability. With this feature, the user is able to serially display a succession of saved graphs. The user reads the list of displays back into the system and then uses the mouse button as a sequencer. Using the mouse in combination with input commands, the user can cycle through the pictures, thus simulating a real interaction sequence with the system. This sequencing feature will be valuable in allowing critiquing of both the user interface display and the visual appearance of a user's session.

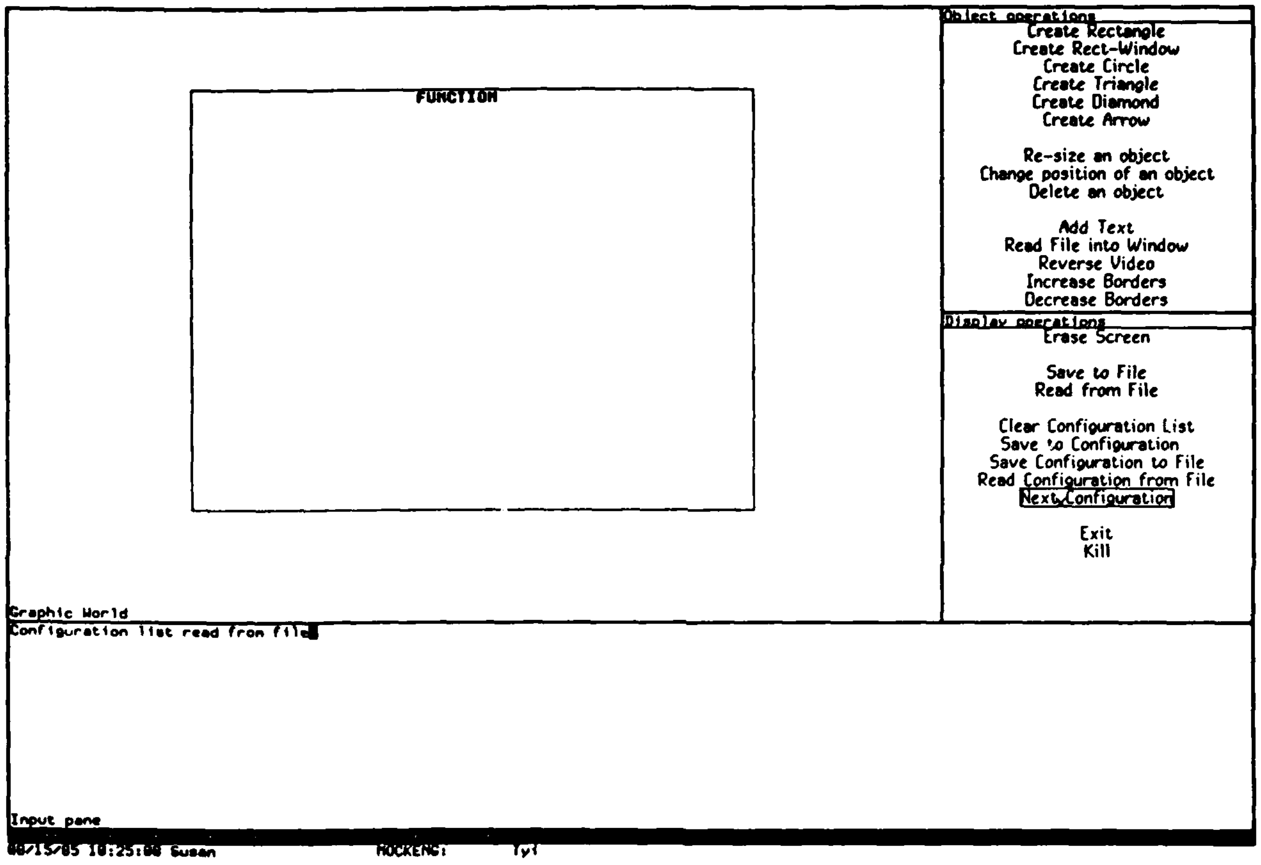
For a more detailed explanation of the screen layout and the available commands, see the appendix.

7.3 Use with PRL/PL

The Mock Engine was used to provide a visual presentation of the manner in which a user would develop a database query using the PRL Picture Language. Each step in the query building process was saved on the configuration list; once all steps had been developed and saved, the entire sequence was saved to a file. An example of the process used in developing the query *"Find all functions containing loops and if-statements"* will be presented in the following four figures. The first figure displays the *FUNCTION* box; the second shows a *LOOP* contained within the *FUNCTION*; the third shows a *FUNCTION* box containing both a *LOOP* and an *IF-STATEMENT*; finally, the *FUNCTION* box is heavily bordered showing that all *functions* matching the query should be returned from the search.

The building of the query can then be simulated using the "configuration sequencing" capability of the Mock Engine. It is often far easier for a viewer to be able to understand and comment on a visual presentation than on strictly textual commentary.

All of the queries discussed in the PRL Picture Language chapter of last year's annual report [Brzustowicz 84] have been developed using the Mock Engine. Besides giving the user a pictorial view of the query development process, this work has also been helpful in showing the system designers which queries could become tedious to develop due to the large number of mouse selections necessary for their creation. A system developer building an actual picture language query mechanism would be able to use this knowledge during development.



```
Configuration list read from file
```

```
08/15/85 10:25:00 Susan                    HOCKENG:      Ty!
```

Figure 7-1: Find all Functions

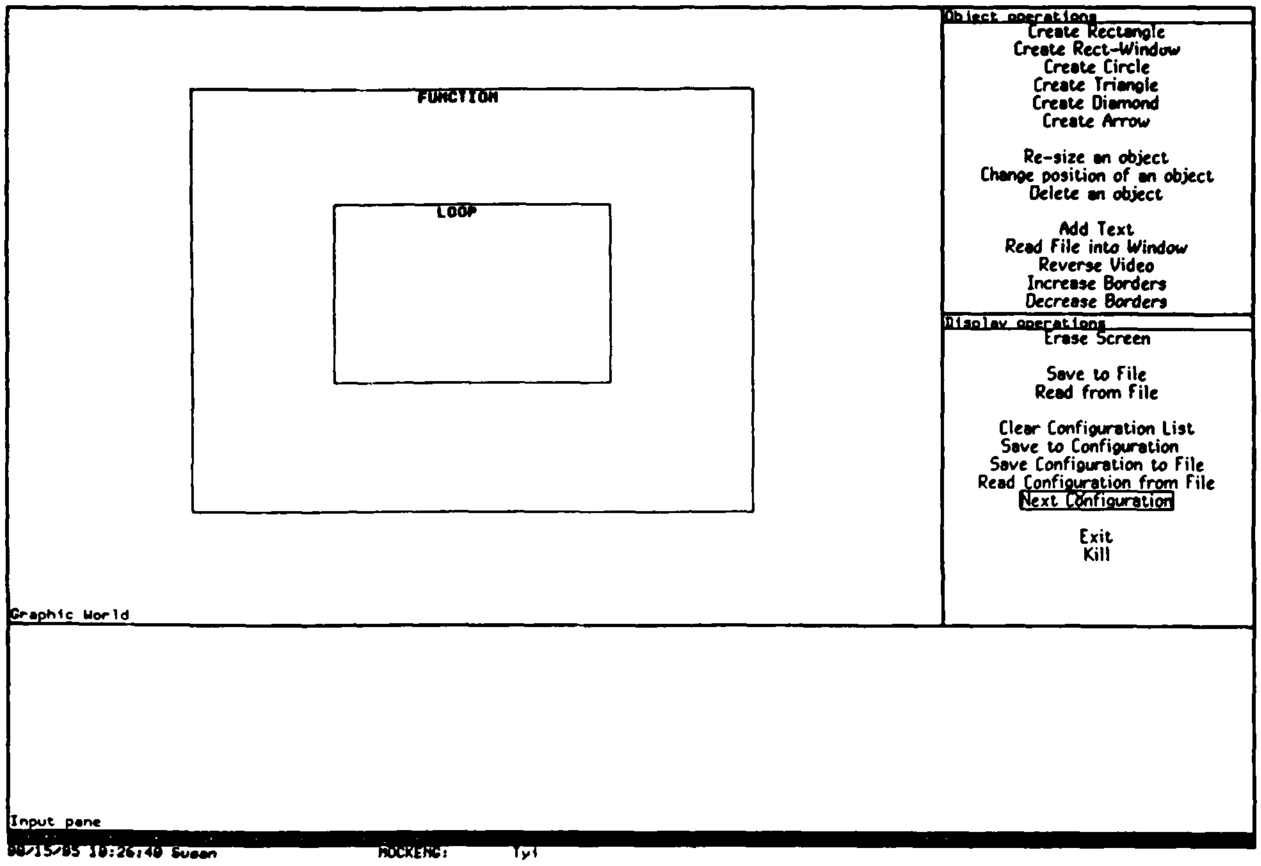
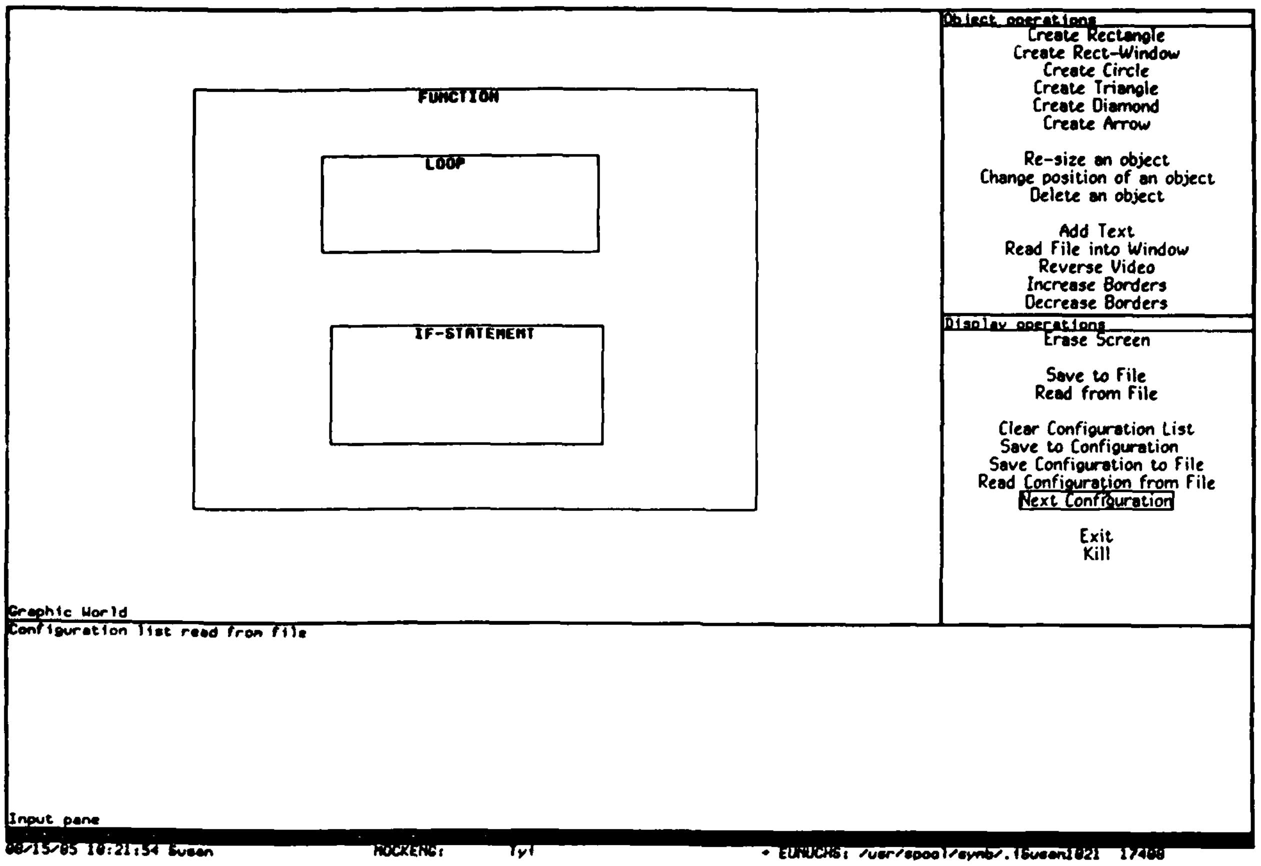


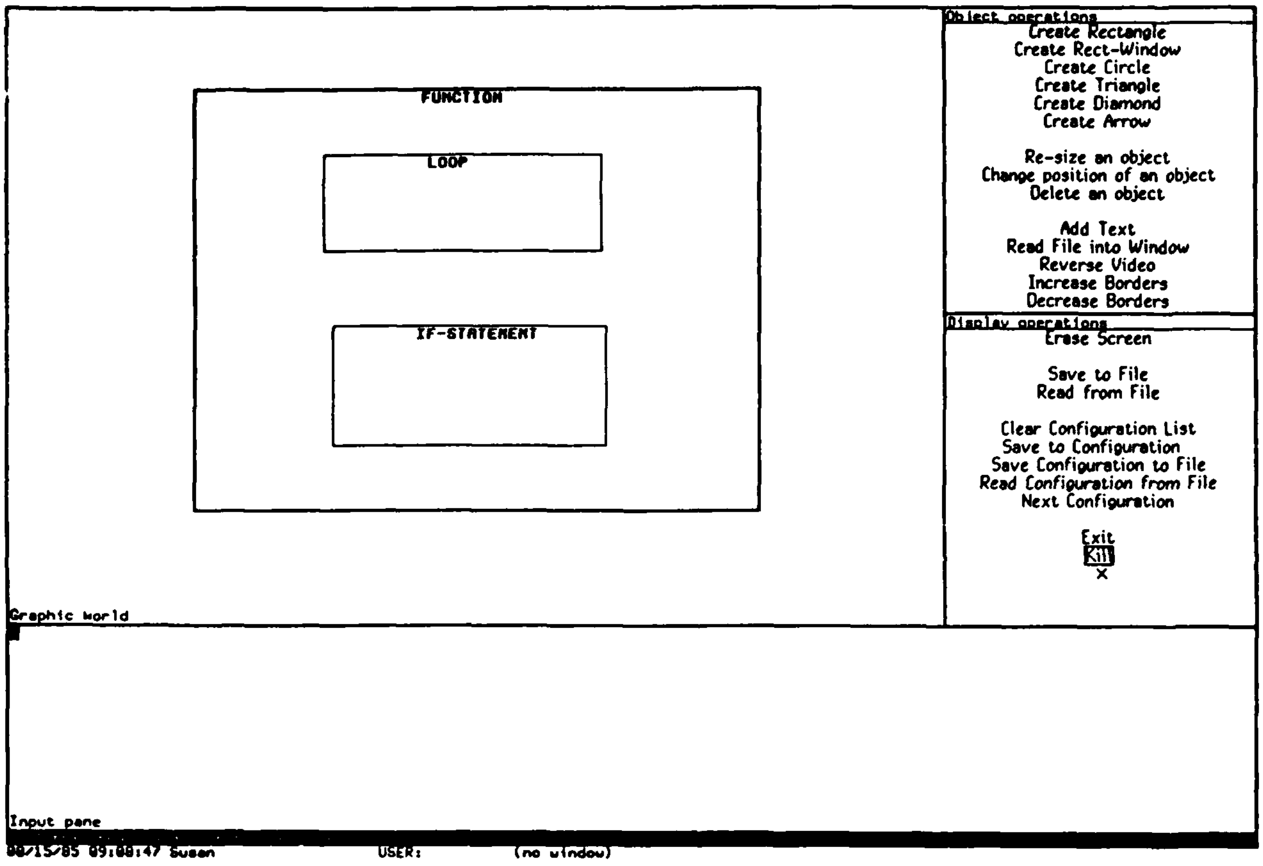
Figure 7-2: Find all Functions containing Loops



```
Configuration list read from file
```

```
08/15/85 10:21:54 Susan          ROCKENG:      Ty!               * EUNUCHS: /usr/spool/synb/.1Susan1021  17400
```

Figure 7-3: Find all Functions containing Loops and If-Statements



```
08/15/85 09:08:47 Susan          USER:          (no window)
```

Figure 7-4: Return all functions matching the query

8. Efficient Retrieval in the IPE

8.1 Searching in the Extended Program Model

Since the purpose of the Program Reference Language is to provide a way to reference programs, one of the key features of a PRL system must be a search facility. The design of the search theory is being based on the proposed PRL Picture Language, a pictorial representation that provides the ability to specify high-level search requests. Using this as a foundation allows the study of the development of basic searching techniques. Even if the PRL Picture Language is not the final query mechanism used in the IPE, the intermediate language into which the Picture Language will be translated for access into the EPM will be similar in nature to the intermediate language that is eventually used in the system. The remainder of this chapter assumes the PRL Picture Language to be the user interface into the EPM.

The PRL research effort is oriented towards providing better methods for program comprehension. Program comprehension becomes of particular importance when dealing with large programs (which may exceed the limitations of what one person can understand). Thus, one of the goals of the PRL research is to provide techniques that will work for large programs.

In dealing with large program search spaces, there are two critical issues: precision and efficiency. The term "precision" refers to the percentage of retrieved items that are actually of interest. As the size of the retrieval space increases, precision becomes more critical. In small search spaces, a small number of false hits are only a minor annoyance to the user; however, if the search space is increased by several orders of magnitude, the corresponding increase in false hits becomes unacceptable.

The PRL inherently simplifies the process of making requests more precise. By allowing the formulation of requests on the basis of various representations (i.e., text, syntax, flow, and/or cliches), requests are easier to formulate *and* they are more specific (and constrained) than simple (text-only) requests.

When the size of the retrieval space increases, the issue of efficiency also becomes more critical. In dealing with a single module, it may be acceptable for the search process to be relatively slow; in dealing with hundreds or even thousands of modules, the time is multipled accordingly, and quickly becomes unacceptable in an interactive programming environment. This section describes how PRL-based searching can be performed efficiently; using the techniques described here, PRL-based search may even be more efficient than conventional searching.

The Intelligent Program Editor (IPE) was originally motivated by the need to provide intelligent support tools for the development and maintenance of medium- to large-scale programs. With small programs, simple retrieval techniques (e.g., text matching) are often sufficient since the requests tend to be simple and the search space small. However, when requests are more complicated, or when the search space is large, these techniques tend to become inefficient and ineffective -- inefficient because they must examine all or most of the search space; ineffective because of the difficulty in making the requests specific enough to return only the desired information.

Through the Extended Program Model (EPM), the IPE provides a much richer vocabulary for retrieval, allowing retrieval by textual, syntactic, and/or limited semantic matching. However, for these techniques to be useful, the IPE must be capable of providing efficient processing for them.

8.2 Intra-Module Query Processing

To explain the PRL search process, it is first necessary to understand the internal representation of programs in the Extended Program Model (EPM). The EPM can be though of as database for storing programs. Its key feature is that it stores programs in a variety of different representations.

The EPM is segmented along two axes (see Figure 8-1). Each module is represented independently in the database (inter-module relationships are represented separately). Within a module, each representation is stored separately; however, there are links between the representations, which provide the basis for converting between representations. Modules provide a natural division from the programming viewpoint; similarly, they provide a natural division from the retrieval perspective. Thus, PRL supports two types of searching: searching within modules and searching between modules. The intra-module search process is the basis for the inter-module search process.

The basic process in retrieval is the application of a PRL query against a single program module. Much of the sophistication of the PRL is based on this capability, which manipulates the multiple representations supported by the EPM in order to handle the full expressiveness of the PRL. The single module retrieval process is currently being researched, and is described here only briefly; for the purpose of explaining inter-module search, the intra-module retrieval process can be treated as a black box.

As currently envisioned, the intra-module search mechanism will handle requests based on the PRL Picture Language. Search requests are limited in their complexity since the PRL/PL restricts the structure and vocabulary of queries. Moreover, the EPM makes several simplifying assumptions to make search easier. As a result, the cost of intra-module searches should be roughly proportional to the size of the module).

A PRL/PL query specifies objects and their relationships; it may also contain boolean connectives. The search mechanism supports the three basic PRL/PL relationships:

- *containment:* structural nesting of objects ("A" is inside "B")
- *precedence:* temporal or spatial ordering of objects ("A" comes before "B")
- *attribute:* characteristics of objects ("A" has a "B")

Containment- and precedence-based searching is similar. All objects to be searched are converted into a common representation (e.g., containment requires all objects to be converted to a syntactic representation); after conversion, a tree traversal is sufficient to identify target objects.

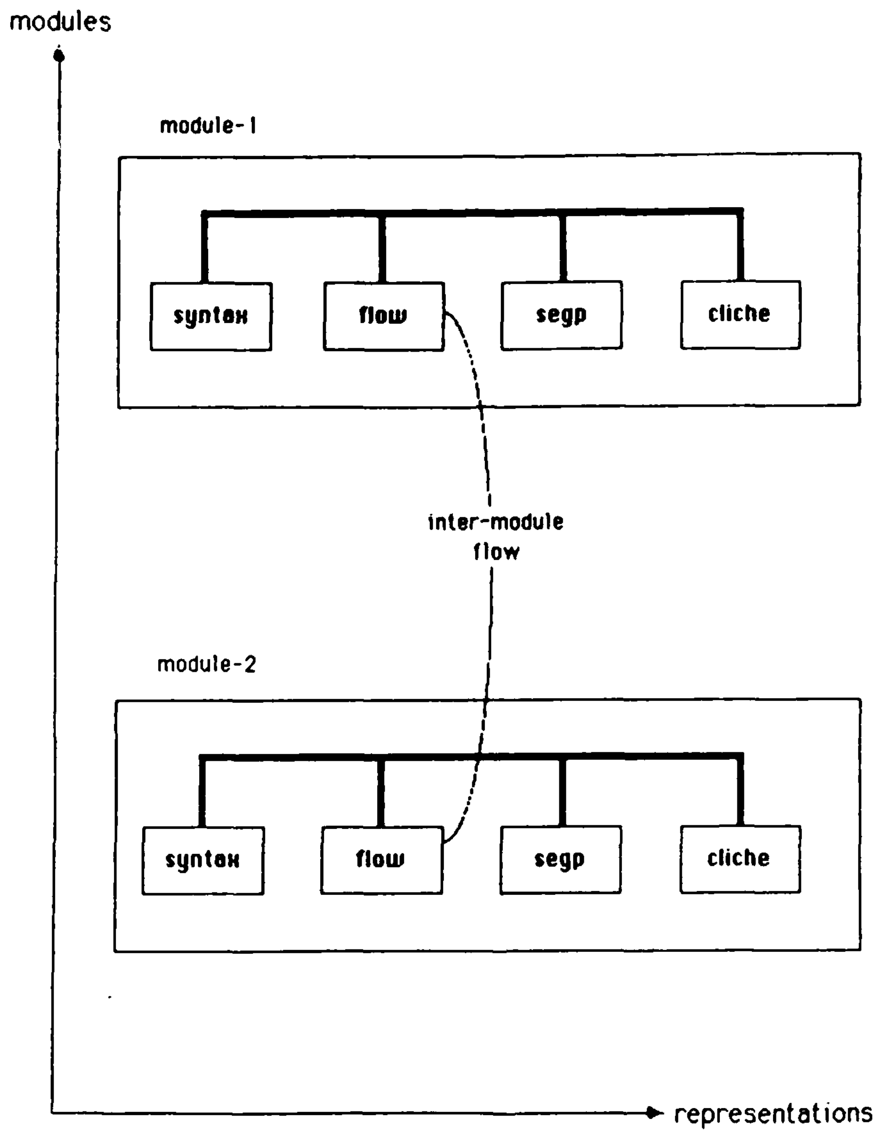


Figure 8-1: Structure of the EPM

Searching based on attributes requires search through the class hierarchy for the appropriate attribute. If the attribute is found, representation conversions and/or computations may be required.

Search may require objects to be converted to different representations. The EPM currently provides one-to-one mappings between representations, simplifying the conversion process. A later version will support fuzzy and one-to-many relationships, possibly requiring searching and computation to do conversion (and thereby increasing the cost of intra-module searching).

8.3 Efficient Inter-Module Query Processing

Inter-module retrieval can be done by repeatedly applying the intra-module search mechanism to each module in the system and concatenating the results. This basic process will be quite inefficient, and not all requests can be satisfied by intra-module searching.

Performing inter-module search by repeated application of the intra-module search process over the entire set of modules is referred to as the *naive* model. It represents an upper bound on the amount of work that a search process must do -- i.e., search everything. In a large application, the naive model becomes prohibitively expensive (and unusable in real-time), since the per module search costs are multiplied by the number of modules.

There are a variety of methods that might be used to reduce the search space. The technique described here is based on indexing, a well-known technique that seems especially suitable for the PRL. Our previous experience with applying indexing techniques indicates that speed improvements (over naive methods) of several orders of magnitude are possible.

8.3.1 Indexing

An index can be thought of as a mapping between the terms of a retrieval request and the target space. Given a request term, an index can be used to locate that term in time proportional to the size of the index, instead of the size of the entire search space. There are two basic types of indices. A *direct* index provides a mapping from modules to terms. It answers questions of the form *"Where does term-1 appear in module-1?"* An *inverted* index provides a mapping from terms to modules. It answers questions of the form *"Which modules contain an term-1?"* Figure 8-2 presents an example.

term	**modules**
case-statement	module-4
goto-statement	----
if-statement	module-1 module-2 module-3 module-5
loop-statement	module-1 module-3 module-5

Figure 8-2: A Simple Inverted Index

The EPM will have several inverted indices, one for each program representation (Figure 8-3). Each inverted index maps from an object (in that representation) to a module. For example, in the syntax inverted index, there is a list of all modules containing *loops*, a list of all modules containing *packages*, etc.

The indices are used to *filter* out modules that cannot possible satisfy a query, before the intra-module search process is invoked on that module. Since filtering is (by design) a much simpler process than searching, considerable speedup can be achieved by eliminating modules from consideration. However, the reduction of the search space depends on the particular terms in a query. If, for example, a query contains terms that are contained in few of the modules, then many modules will be eliminated, and a

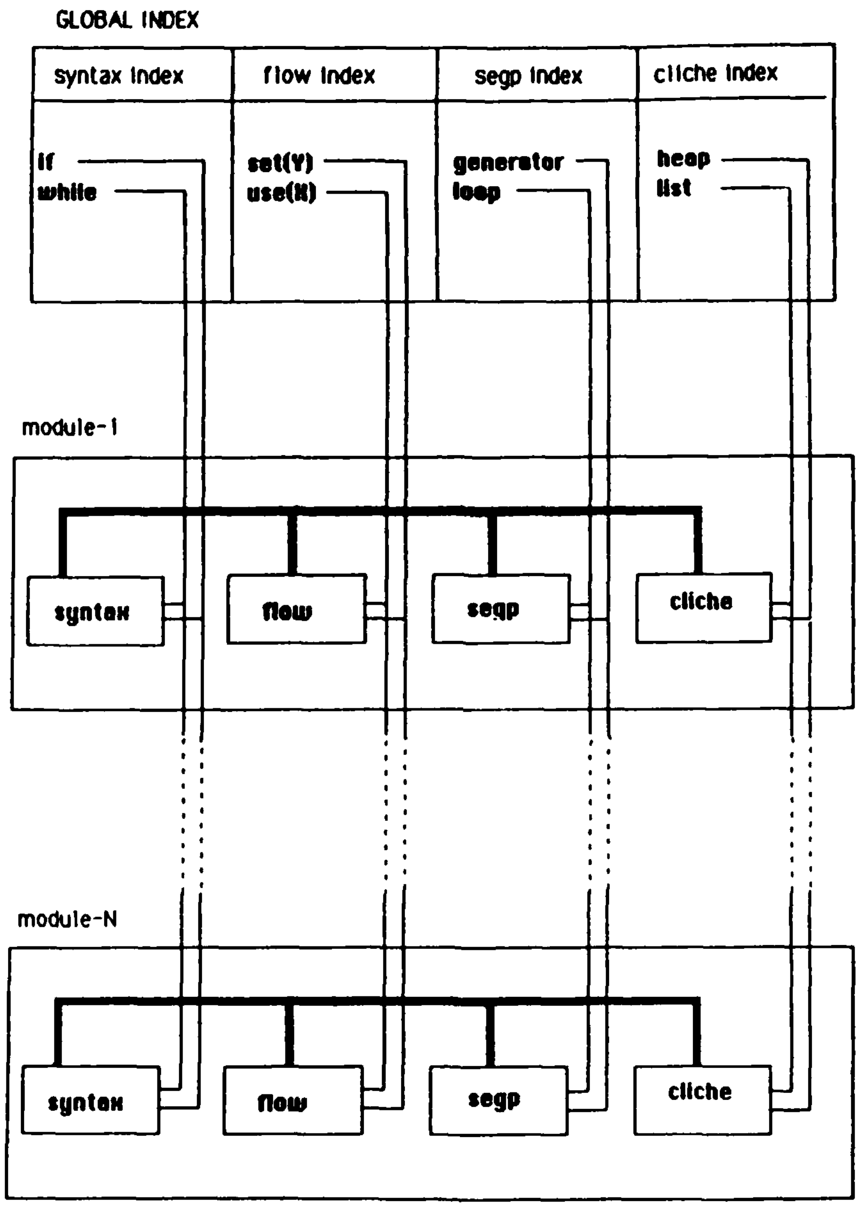


Figure 8-3: Inverted Indices in the EPM

significant speedup will be realized; on the other hand, if the terms are common, the filtering process may actually cause the retrieval process to take longer than a naive query process.

The technique used for filtering modules can be thought of as a stripped down version of intra-module query evaluation algorithm. It converts the query tree into an AND/OR tree. The AND/OR tree represents the minimal requirements a module must satisfy for the full query to succeed.

The AND/OR tree is constructed by converting all relations (containment, precedence, negation, quantification) into AND or OR relationships. If the query requires the presence of some object (e.g., the query *"find an A which contains a B"* requires both A and B), then the conjunction of A and B is added to the AND/OR tree. If an object is not required (e.g., the query *"find an A which does not contain a B"* does not require B), then it is omitted from the tree. Figure 8-4 shows an example of a PRL/PL query, its translation to tree form, and the resulting AND/OR tree.

To determine if a module is a candidate for full searching, the AND/OR tree is evaluated with respect to the indices for that module. Only modules that evaluate to *true* are candidates for the full query evaluation; the other modules are missing necessary components. Thus, the filtering algorithm can be thought of as "structureless" query processing -- all of the components of the original query must be present, but their relationships are irrelevant.

Unless the filtering process actually eliminates some modules, the cost in this case would actually be higher than the naive case. In a large system, it is expected that filtering will in fact reduce the number of modules. A technique for computing the break even point for the filtering process (i.e., the number of modules that have to be eliminated for the filtering to be cost effective) remains to be determined.

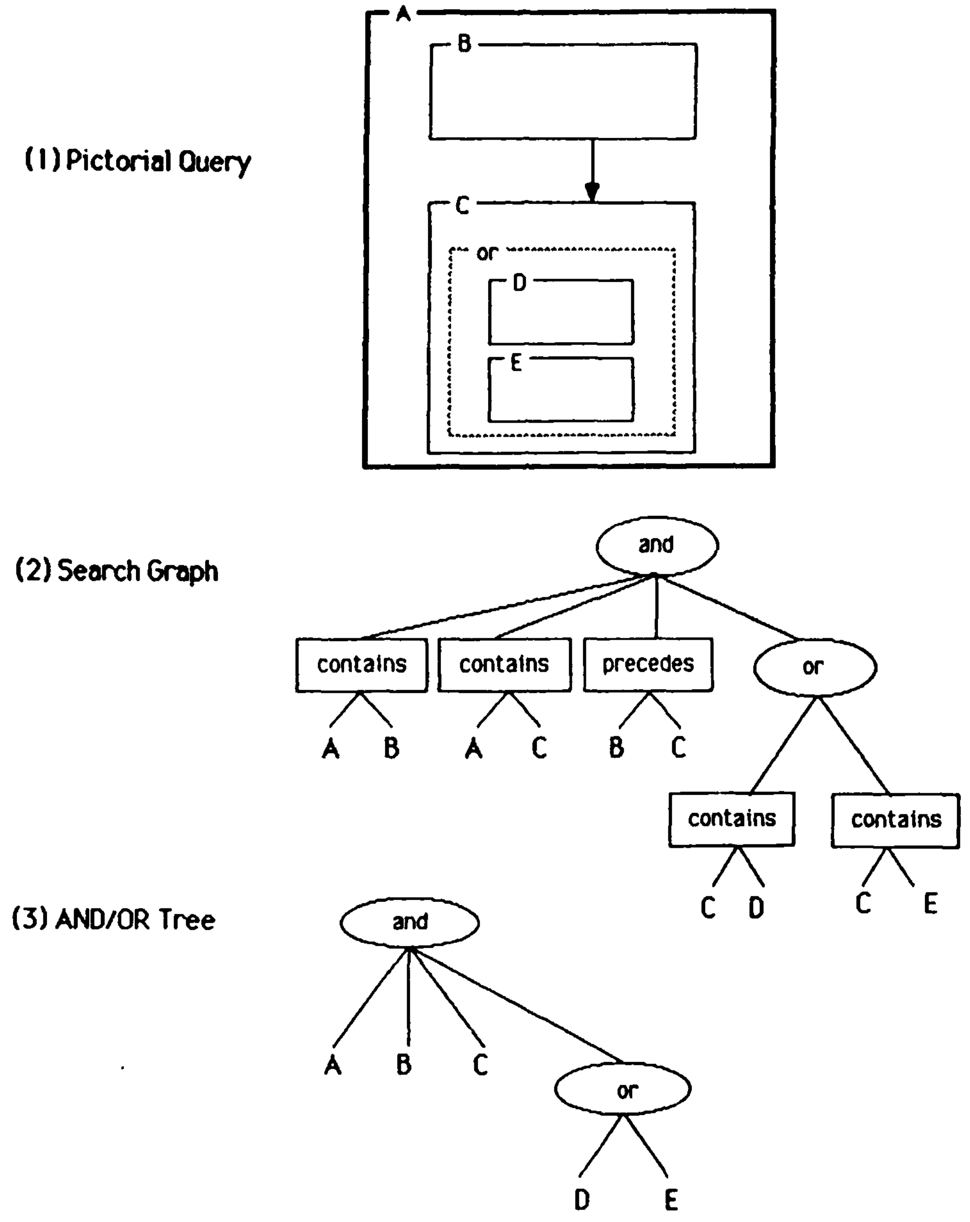


Figure 8-4: Mapping from a PRL/PL Query to an AND/OR Tree

8.3.2 Filtering

The filtering process operates on the inverted indices (and not the modules themselves). The process can be performed either sequentially (evaluating the AND/OR tree once for each module) or in parallel (evaluating the AND/OR tree once for all modules). From an efficiency viewpoint, there are different advantages to both; selecting a particular method might best be done empirically.

The parallel evaluation technique evaluates all modules at each node in the AND/OR tree. A bit vector is used to represent the modules, with one bit indicating the status of each module. At the end of the evaluation, the bit vector indicates which modules passed through the filter. For each operation in the AND/OR tree, if there are M modules in the database, M boolean operations are performed on the bit vector. Parallel evaluation allows no shortcuts: if there are N operations in the tree, O(M * N) boolean operations must be performed. However, bit vectors are compact, and can be executed quite efficiently by most computers, so the total overhead is reasonable.

The sequential evaluation technique accesses the tree once for each module (and accessing the set of indices about that often). The sequential process permits short-circuit boolean evaluation, allowing modules to be eliminated without necessarily evaluating the entire tree. It can also make use of frequency information to reorder the tree into a more optimal form, allowing early evaluation of those indexed items with the highest discrimination ability. Thus, sequential evaluation is much more sensitive to the data; determining the associated cost will require experimentation.

It might be possible to perform a pre-filter process, to reduce the cost of the filtering process itself. The pre-filter process could be an even more reduced version of the filtering process, or it could be based on statistical information about previous filterings.

8.3.3 Minimizing Index Size

Non-indexed retrieval schemes have the advantage of requiring no extra overhead ɔ support retrieval; retrieval can be done directly on the modules themselves. Indexing schemes have the overhead of creating and maintaining indices. To prevent the cost of the overhead from exceeding the savings achieved by indexing, there are several ways of reducing indexing overhead.

8.3.3.1 Simplifying the Index

In our indexed retrieval model, only a simple indexing structure is needed. Since the index is used to answer true/false questions about each module, the index needs only an indication of whether an object is contained in a module. It is not necessary to specify where in the module it appears.

This greatly simplifies index maintenance, since it is only necessary to update the index when a new object is added to a module or when the last of an object is removed. It also reduces the size of the index, since it is only necessary to store at most one indicator per object/module pair.

Moreover, index maintenance in an environment like the IPE can be done incrementally, as programs are modified. This can even be done in the background, invisible to the user.

8.3.3.2 Limiting the Index

The vocabulary of the PRL contains all objects represented by the IPE; this includes all representation levels (text, syntax, semantics, and documentation). A simplified indexing scheme would require all terms in the vocabulary to be indexed. However, the indexing scheme proposed here does not require that all terms be indexed; doing so might actually be detrimental to efficiency.

The purpose of the index is to filter out modules that cannot possibly contain the sought objects. Thus, an object that occurs in every module is useless with respect to indexing, since it will never cause a module to be eliminated. An object that appears in most modules would also be less useful as an index term, since it too would be a poor filter. For an object occurring with great frequency, it may be less work to assume that the object is in all modules than to actually perform filtering based on that term.

On the other hand, objects that occur with too low a frequency may not be worth the expense to index, especially if there are a large number of this type of object, since these objects would be rarely specified in search requests. The best example of this is in at the textual level, where it is infeasible to maintain an index of all possible substrings.

The decision about what to index and what not to index can be made at design time, when the indexing mechanisms are being constructed, or at run time, when frequency information is available. The best strategy might be to make these decisions at both times. For example, it can be decided a priori that maintaining indices for the textual representation is unlikely to be useful; however, deciding which syntactic features should be indexed would be best determined a posteriori, using frequency information.

8.4 Joining

The search model described earlier was based on the assumption that all queries could be satisfied by examining a single module at a time. In fact, this limitation prevents the general use of meta-variables, which provide a means for joining arbitrary components in the program space. However, our basic model can be extended to handle meta-variables.

The algorithm for processing a query with meta-variables operates by taking the query and decomposing it into subqueries without meta-variables. All of the subqueries are then processed as regular queries, using the previously described indexed retrieval algorithm. Then, the results are joined together, which returns the final result.

Queries are *decomposed* in joinless queries in two steps:

1. *find lowest common ancestors:* For every pair of meta-variables, locate the lowest common ancestor in the tree; this node is referred to as the *join node* (Figure 8-5-1). A join node may actually join any number of meta-variables.

2. *break off branches:* At each join node, break off all branches that contain

meta-variables (Figure 8-5-2). The resulting branches (including what is left of the original tree, if anything) are the new joinless queries (i.e., no branch has more than one meta-variable).

This set of subqueries is then processed in the standard fashion (i.e., including filtering). Subqueries containing single meta-variables are handled by letting the meta-variables match anything satisfying specified constraints.

Each subquery will have produced a set of values as a result. These results are then *combined* as follows:

1. *rebuild tree:* Associate the resulting set of values with the top node of each branch, and put the entire tree back together (Figure 8-5-3).

2. *join results:* The intermediate results belonging to the join nodes can now be computed by performing an operation on all the result sets of the children (Figure 8-5-4). The specific operation depends on the type of the join node. Specifics are presented in figure 8-6.

The *equi-join* and *Cartesian product* operators are relational algebra operators. Each branch of the divided query produces a set of tuples as a result. Equi-join merges two branches by combining pairs of tuples (one tuple from each branch) that have matching meta-variables. Cartesian product merges two branches by producing all possible pairwise combinations of tuples. Operations for other join nodes are the functional composition of the node relation with Cartesian product.

Joining can be an expensive operation, especially when the number of objects to be joined is large, as can happen when the join is applied across the entire program space. If it can be determined that a join operation is constrained to occur within a single module, then the join can be performed on a per-module basis, as part of the intra-module search process.

8.5 Summary

Due to the sophistication of the PRL and EPM, we originally expected PRL-based search to be a slow and inefficient process; we now believe that the PRL may be able to provide efficient searching across large programs. Moreover, because the EPM provides

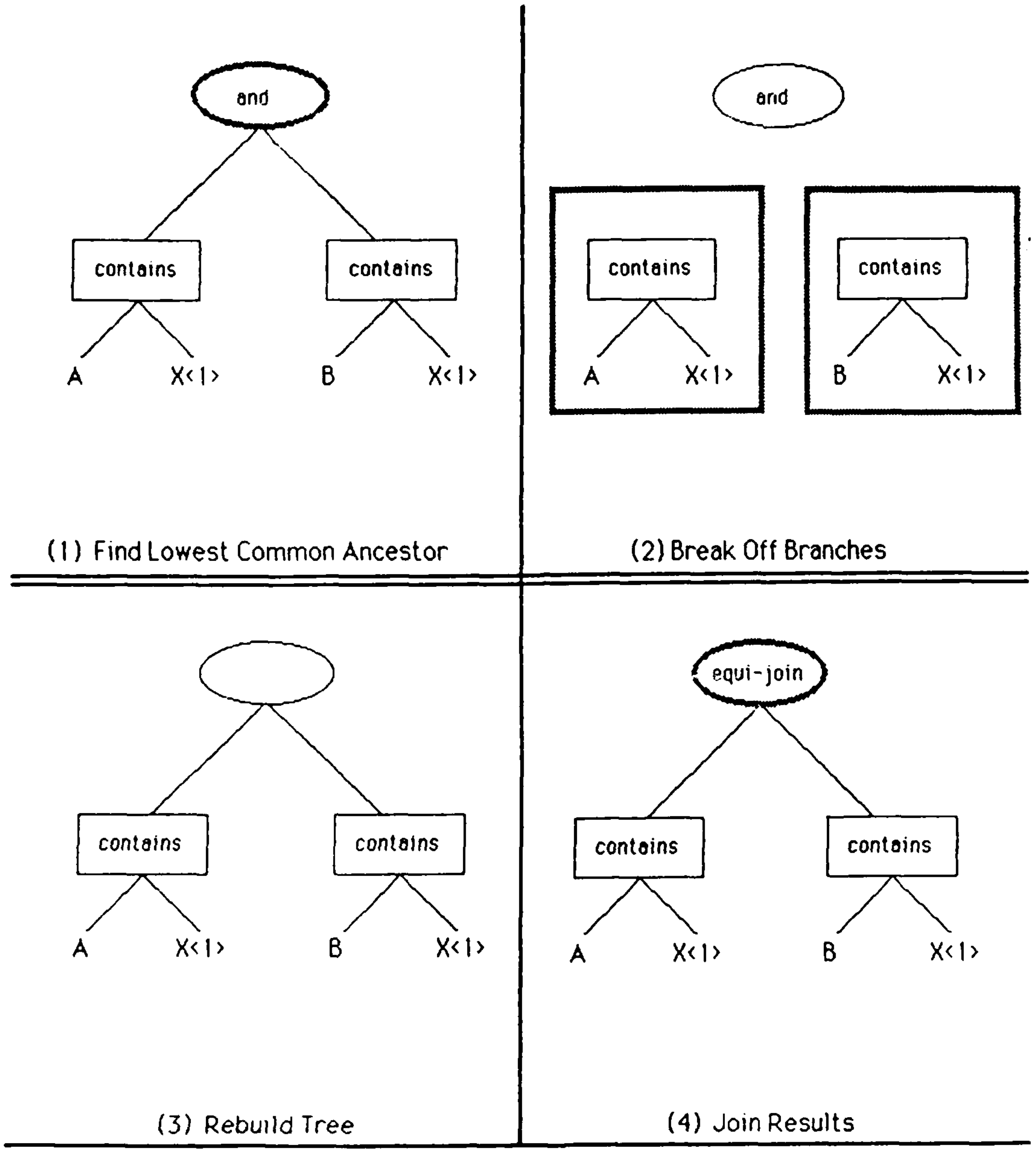


Figure 8-5: Handling Joins in Searches

type of join node	*operation*
and	equi-join
or	Cartesian product
relation	*relation* ∘ Cartesian product

Figure 8-6: Joining Operations

a natural and well-defined basis for indexing -- the multiple program representations -- it may turn out that PRL-based searching can be performed more efficiently than conventional searching.

A. Example Queries in the Picture Language

This appendix contains the queries delineated in [Domeshek 84]. Each query is expressed in three different representation languages: Predicate Calculus, LISP pseudo-code, and the formal Picture Language.

Generally, examples will be stated in four steps. First the query is expressed in Predicate Calculus. That formal specification is converted into LISP pseudo-code. The pseudo-code is converted into PL and simplified. Finally an abstract example of the results is presented.

A.1 Top Level Schematic For The Examples

1. LOGICAL SENTENCE:

   The query expressed in Relational Calculus.

   Extended logic refers to the Annotated Predicate Calculus of R. Michalski.

   Note that the top level of description, (i.e.: All OBJ such that ...), is the universal access to the database.

2. COMPUTABLE PROGRAM:

   The LISP pseudo-code representation of the logical sentence.

   The implementation is in terms of recognizers and collectors. An encapsulating

   ```
   (loop over all objects ...
         return list of successes)
   ```

   is assumed for all code.

   The function (RECOG x) looks like:

   ```
   (= (type OBJ) 'X)
   ```

   Search functions will return a list of objects meeting the specification and their contents or containers. The additional information can be removed later if necessary, at the interface level.

3. PL STRUCTURE:

The derivation of the Picture Language representation from the LISP pseudo-code.

A name suffixed by an "x" is a quantifier label.

(type x 'y) is expressed merely as y.

The quantified version is formed by skolemizing and eliminating quantifiers.

Skolem functions are expressed as Sobj.

Finally, a mnemonic for Sobj is reintroduced, eg: lpe. The "e" is for existential.

The PL form is a pattern-matching template for LISP programs. For example, in Query #1:

```
[ fn $a [ lp $b ] $c ]
```

matches LISP

```
( fn any1 ( lp any2 ) any3 )
```

The $ variables match any structures including nil. The abstract "fn" matches any *true* return from (recog FN); "lp" matches any (recog LP), such as "FOR" "WHILE" etc. Finally, brackets match parentheses.

4. RESULT

An abstract representation of what the output should look like.

A.2 The Query Examples

1. FIND THE FUNCTIONS THAT CONTAIN LOOPS.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exists X . (type X 'LP) and (contains OBJ X)
```

PROGRAM:

```
(if (and (recog FN)
         (loop over contents-of-FN
            (if (recog LP)
                (collect LP into LPs)) ))
    (collect-and-return (list FN LPs)) )
```

Note that all LPs are collected in order to keep the symmetry
of CONTAINS. Alternatively, non-symmetrical code would be:

```
(if (and (recog FN)
         (loop over contents-of-FN
            (if (recog LP)
                (return t)) ))
    (collect-and-return FNs) )
```

PL:

```
[ OBJx [ (type obj 'fn) Xx (type x 'lp) (contains obj x) ]]

==>  [ FNx [ LPx (contains fn lp) ]]
==>  (contains fn Sfn)
==>  [fn [lpe]]
```

DISCUSSION:

See NOTE A.

RESULT:

```
( (FN1 (LP1 LP2))
  (FN2 (LP3 LP4 LP5))
  (FN3 (LP6))
        ... )
```

2. FIND THE LOOPS CONTAINED IN FUNCTIONS.

LOGIC:

```
All OBJ . (type OBJ 'LP) and
  Exists X . (type X 'FN) and (contains X OBJ)
```

PROGRAM:

Same as #1

The directionality of containment is irrelevant at this level.
The interface level can take care of this distinction.

PL:

```
[ OBJx [ (type obj 'lp) Xx (type x 'fn) (contains x obj) ]]

==>   [ LPx [ FNx (contains fn [illegible] ]]
==>   (contains Slp lp)
==>   [fne [lp]]
```

RESULT:

```
(  (LP1 FN1)
   (LP2 FN1)
   (LP3 FN2)
   ... )
```

3. FIND ALL FUNCTIONS CONTAINING LOOPS AND IF-STATEMENTS.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exists X Y . (type X 'LP) and (type Y 'IF)
               and (contains OBJ X)
               and (contains OBJ Y)
```

EXTENDED LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exists X Y . (type X 'LP) and (type Y 'IF)
               and (contains OBJ (X and Y))
```

PROGRAM:

```
(if (and (recog FN)
         (loop over contents-of-FN
            do
            (if (recog LP)
                (collect into LPs))
            (if (recog IF)
                (collect into IFs)))
         (not-empty LPs)
         (not-empty IFs) )
    (collect-and-return (list FN LPs IFs)) )
```

PL:

```
[ OBJx [ (type obj 'fn) Xx Yx (type x 'lp) (type y 'if)
         (contains obj x) (contains obj y) ]]

==>  [ FNx [ LPx IFx (contains fn lp) (contains fn if) ]]
==>  (contains fn S1fn) (contains fn S2fn)
==>  [fn [lpe]] [fn [ife]]
```

Distribution converts this to the extended logic form:

```
==>  [fn [lpe ife]]
```

RESULT:

```
(  (FN1 (LP1 LP2) (IF1 IF2 IF3))
   (FN2 (LP3) (IF4 IF5))
   ... )
```

4. FIND ALL FUNCTIONS CONTAINING A LOOP FOLLOWED BY AN IF-STATEMENT.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exists X Y . (type X 'LP) and (type Y 'IF)
               and (contains OBJ X)
               and (contains OBJ Y)
               and (follows Y X)
```

PROGRAM:

```
(if (and (recog FN)
         (loop over contents-of-FN
            (if (and (recog LP)
                     (loop over remaining-contents-of-FN
                        (if (recog IF) (save IF)) ))
                (collect into LPs-with-IF))))
    (collect-and-return (list FN LPs-with-IF)))
```

Note: the "followed by" is assumed to be implicit in the sequence of the contents-of-FN. That is, if X contains Y and Z follows Y, then X contains Z.

PL:

```
[ OBJx [ (type obj 'fn) Xx Yx (type x 'lp) (type y 'if)
         (contains obj x) (contains obj y) (follows y x) ]]

==>  [ FNx [ LPx IFx (contains fn lp) (contains fn if)
                     (follows if lp) ]]
==>  (contains fn S1fn) (contains fn S2fn)
                        (follows S1fn S2fn)
==>  [fn [lpe]] [fn [ife]]
         \__________/
```

which can be rewritten:

```
[fn [lpe-->ife]]
```

RESULT:

```
(  (FN1 (LP1 IF1))
   (FN2 (LP2 IF2) (LP3 IF3))
   ... )
```

5. FIND ALL FUNCTIONS WHICH CONTAIN LOOPS THAT CONTAIN IF-STATEMENTS.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exists X Y . (type X 'LP) and (type Y 'IF)
                 and (contains OBJ X)
                 and (contains X Y)
```

PROGRAM:

```
(if (and (recog FN)
         (loop over contents-of-FN
            (if (and (recog LP)
                     (loop over contents-of-LP
                        (if (recog IF)
                            (collect into IFs))))
                  (collect into LPs-with-contained-IFs))))
    (collect-and-return (list FN LPs-with-contained-IFs)))
```

PL:

```
[ OBJx [ (type obj 'fn) Xx Yx (type x 'lp) (type y 'if)
         (contains obj x) (contains x y) ]]

==>  [ FNx [ LPx IFx (contains fn lp) (contains lp if) ]]
==>  (contains fn S1fn) (contains S1fn S2fn)
==>  [fn [lpe]] [lpe [ife]]
```

which could be rewritten:

```
[ [[fn] [lpe]] [lpe ife] ]
```

RESULT:

```
(  (FN1 (LP1 (IF1 IF2)) (LP2 (IF3)))
   (FN2 (LP3 (IF4)))
   ... )
```

6. FIND THE FUNCTION NAMED BAR.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
          (slot-value (slot-type OBJ 'name) 'BAR)
```

PROGRAM:

```
(if (and (recog FN)
         (= (slot-value FN 'name) 'BAR))
           (return (representation FN)))
```

PL:

```
[ OBJx [ (type obj 'fn)
         (slot-value (slot-type obj 'name) 'BAR) ]]

==>  (slot-value (slot-type fn 'name) 'BAR)
```

REPRESENTATION:

function
name: BAR

The above configuration is seen as a single descriptor.
Since the NAME is a fundamental reference, an abbreviation might be appropriate, for example, fn:bar

RESULT:

Some representation of BAR.

The representation may be the code typographical representation, the frame with slots filled in, or the location of BAR relative to a context of inquiry (callers, used variables, etc.)

7. FIND ALL FUNCTIONS THAT USE THE VARIABLE FOO.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exist X . (type X 'VAR) and
            (slot-value (slot-type X 'name) 'FOO)
            and (contains OBJ X) and (uses OBJ X)
```

PROGRAM:

```
(if (and (recog FN)
         (loop of contents-of-FN
                    (if (and (recog VAR)
                     (= (slot-value VAR name) 'FOO)
                     (uses FN VAR) )
               (collect VAR) ))
    (collect-and-return (list FN VAR)) )
```

PL:

```
[ OBJx [ (type obj 'fn) Xx (type x 'var)
         (slot-value (slot-type x 'name) 'foo)
         (contains obj x) ]]

==>  [ FNx [ VARx (slot-value (slot-type var 'name) 'foo)
                   (contains fn var) ]]
==>  (slot-value (slot-type Sfn 'name) 'foo)
        (contains fn Sfn)
==>  [fn [var:foo]]
```

REPRESENTATION:

variable
name: FOO

or just var:foo

RESULT:

```
( (FN1 FOO)
  (FN2 FOO)
  (FN3 FOO)
  ... )
```

NOTES:

USE implies CONTAINS.
The variable FOO = variable named FOO.
Different VARs can have the same name.
The concept USES needs a binding-context, or focus, to be useful.
A request for a variable (which changes) named X must be syntactic.

8. FIND ALL FUNCTIONS CONTAINING LOOPS OR IF-STATEMENTS.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exists X Y . (type X 'LP) and (type Y 'IF)
               and { (contains OBJ X) or (contains OBJ Y) }
```

EXTENDED LOGIC:

```
 ... (contains OBJ (X or Y))
```

PROGRAM:

```
(if (and (recog FN)
         (loop over contents-of-FN
            (if (or (recog LP)
                    (recog IF))
                (collect into LPs-IFs))))
    (collect-and-return (list FN LPs-IFs)))
```

PL:

```
[ OBJx [ (type obj 'fn) Xx Yx (type x 'lp) (type y 'if)
         [ [(contains obj x)] [(contains obj y)] ]  ]]

==>  [ FNx [LPx IFx [[(contains fn lp)]
                     [(contains fn if)]] ]]
==>  [[ [[(contains fn S1fn)] [(contains fn S2fn)]] ]]
==>  [ [   [fn [lpe]]    ] [   [fn [ife]]    ] ]
==>  [ fn [lpe] [ife] ]
```

RESULT:

```
(  (FN1 (LP1 LP2 IF1 LP3))
   (FN2 (IF2))
   (FN3 (IF3 LP4))
   ... )
```

9. FIND ALL FUNCTIONS CONTAINING LOOPS AND IF-STATEMENTS.

Same as #3.

Note: The explicit AND box is unnecessary.

10. FIND ALL FUNCTIONS CONTAINING LOOPS OR IF-STATEMENTS.

Same as #8.

The explicit OR box is unnecessary.

11. FIND THE FUNCTIONS THAT DO NOT CONTAIN LOOPS.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  not Exists X . (type X 'LP) and (contain OBJ X)
```

PROGRAM:

```
(if (and (recog FN)
         (loop over contents-of-FN
            (if (recog LP)
                (return nil))
            (finally return t) ))
    (collect-and-return FNs) )
```

PL:

```
[ OBJx [ (type obj 'fn)
         [ Xx (type x 'lp) (contains obj x) ]  ]]

==>  [ FNx [ [ LPx (contains fn lp) ] ]]
==>  [ FNx LPx (contains fn lp) ]
==>  [ (contains fn lp) ]
==>  [ [fn [lp]] ]
```

DISCUSSION:

See NOTE B.

RESULT:

```
(FN1 FN2 FN3 ...)
```

12. FIND THE IF-STATEMENTS NOT CONTAINED IN LOOPS.

LOGIC:

```
    All OBJ . (type OBJ 'IF) and
      Not Exist X . (type X 'LP) and (contains X OBJ)
```

PROGRAM:

Assume each IF has only one location of its representation;
the program requires two passes through OBJs.

Pass 1:

```
    (if (recog IF)
        (collect into IFs))
```

Pass 2:

```
    (if (recog LP)
        (loop over contents-of-LP
           (if (recog IF)
               (remove IF from IFs) ))
    (finally return remaining-IFs)
```

PL:

```
    [ OBJx [ (type obj 'if) [ Xx  (type x 'lp)
                                  (contains x obj) ] ]]

    ==>  [ IFx [ [ LPx  (contains lp if) ] ]]
    ==>  [ IFx LPx (contains lp if) ]
    ==>  [ [lp [if]] ]
```

RESULT:

```
    (IF1 IF2 IF3 ...)
```

DISCUSSION:

See NOTE C.

13. FIND THE FUNCTIONS WHICH HAVE A LOOP NOT FOLLOWED BY AN IF-STATEMENT.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exists X . (type X 'LP) and (contains OBJ X) and
    if Exists Y . (type Y 'IF) and (contains OBJ Y)
    then not (follows Y X)
```

PROGRAM:

```
(if (and (recog FN)
         (loop over contents-of-FN
            (if (recog LP)
                (if (loop over remaining-contents-of-FN
                          (if (recog IF)
                              (return nil))
                          (finally return t) )
                        (collect into LPs) ))))
    (collect-and-return (list FN LPs)) )
```

PL:

```
[ OBJx [ (type obj 'fn) Xx (type x 'lp) (contains obj x)
         [ Yx (type y 'if) (contains obj y)
           [ [ (follows y x) ]  ]] ]]

==>  [ FNx [ LPx (contains fn lp)
               [ IFx (contains fn if)
                     [[ (follows if lp) ]] ] ] ]
==>  [[ (contains fn Sfn)
        [(contains fn if)(follows if Sfn)] ]]
==>  [fn [lpe]] [ [fn [if]] lpe-->if ]
```

RESULT:

```
(  (FN1 LP1 LP2)
   (FN2 LP3)
   ... )
```

14. FIND THE FUNCTIONS IN WHICH ALL LOOPS CONTAIN IF-STATEMENTS.

That is, find the functions that contain loops such that every contained loop contains an if-statement. See NOTE D.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  All X . (type X 'LP) and (contains OBJ X) and
    Exist Y . (type Y 'IF) and (contains X Y)
```

PROGRAM:

```
(if (and (recog FN)
         (loop over contents-of-FN
            (if (and (recog LP)
                     (loop over contents-of-LP
                        (if (recog IF)
                            (collect into IFs)) ))
                (collect into LPs+IFs)) ))
    (collect-and-return (list FN LPs+IFs)) )
```

PL:

```
[ OBJx [ (type obj 'fn)
         [ Xx [ (type x 'lp) (contains obj x)
                Yx (type y 'if) (contains x y) ]] ]]

==>  [ FNx [[ LPx
              [ (contains fn lp) IFx (contains lp if) ]] ]]
==>  [ [ (contains fn lp) (contains lp Sfnlp) ]]
==>  [fn [lp]] [lp [ife]]
```

RESULT:

```
(  (FN1 (LP1 IF1) (LP2 IF2 IF3))
   (FN2 (LP3 IF4 IF5))
   ... )
```

15. FIND THE FUNCTIONS WHICH CONTAIN 2 LOOPS THAT CONTAIN IF-STATEMENTS.

Note: cardinality can be treated the same as any quantifier.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exist X . (type X 'LP) and (cardinality X 2)
                          and (contains OBJ X) and
    Exist Y . (type Y 'IF) and (contains X Y)
```

PROGRAM:

```
(if (and (recog FN)
         (loop over contents-of-FN
            (if (and (recog LP)
                     (loop over contents-of-LP
                        (if (recog IF)
                            (collect into IFs)) ))
                (collect into LPs+IFs)) )
         (= (number-of LPs+IFs) 2) )
    (collect-and-return (list FN LPs+IFs)) )
```

PL:

```
[ OBJx [ (type obj 'fn)
         Xx (type x 'lp) (cardinality x 2) (contains obj x)
         Yx (type y 'if) (contains x y) ]]
==>  [ FNx [ LPx(2) (contains fn lp)
             IFx (contains lp if) ]]
==>  (contains fn lpe2) (contains lpe2 ife)
==>  [fn [lpe2]] [lpe2 [ife]]
```

RESULT:

```
( (FN1 (LP1 IF1) (LP2 IF2 IF3))
  (FN2 (LP3 IF4) (LP4 IF5))
  ... )
```

16. FIND THE FUNCTIONS WHICH CONTAIN AN IF-STATEMENT NOT CONTAINED IN A LOOP.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exist X . (type X 'IF) and (contains OBJ X) and
    not Exist Y . (type Y 'LP) and (contains OBJ Y)
                               and (contains Y X)
```

PROGRAM:

```
(if (and (recog FN)
         (loop over contents-of-FN
            (if (recog IF)
                (collect into IFs)) ))
    (collect-and-return (list FN IFs)) )
```

PL:

```
[ OBJx [ (type obj 'fn) Xx (type x 'if) (contains obj x)
         [ Yx (type y 'lp) (contains obj y)
                           (contains y x) ] ]]

==>  [ FNx [ IFx (contains fn if)
               [ LPx (contains fn lp) (contains lp if) ] ]]
==>  (contains fn Sfn)
       [ (contains fn lp) (contains lp Sfn) ]
==>  [fn [ife]] [ [fn [lp]] [lp [ife]] ]
```

NOTE:

For #16 to be a meaningful query, the semantics of CONTAINS must include the idea of deep and shallow containment.

The PL example is a statement of the intransitivity of containment. Abstractly:

[a [b]] & [b [c]] imply [[a [c]]]

RESULT:

```
(  (FN1 (IF1 IF2))
   (FN2 (IF3))
   ... )
```

17. FIND THOSE FUNCTIONS THAT USE SOME VARIABLE BEFORE SETTING THAT VARIABLE.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exist X . (type X 'VAR) and (uses OBJ X)
                          and (sets OBJ X)
                          and (follows set use)
```

DISCUSSION:

See query #18.

18. FIND THOSE FUNCTIONS IN WHICH A VARIABLE IS NOT SET BEFORE THAT VARIABLE IS USED.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exist X . (type X 'VAR) and (uses OBJ X) and
    if (sets OBJ X) then (follows set use)
```

DISCUSSION:

The FOLLOWS construct has events rather than objects as arguments. Assuming that the events can be objectified, and that FOLLOWS can be dynamic, these translations occur:

PROGRAM:

```
(if (and (recog FN)
         (loop over contents-of-FN
            (if (and (recog VAR)
                     (uses FN VAR)
                     (if (and (sets FN VAR)
                              (follows set use) )
                         (collect VAR into VARs) ))) ))
    (collect-and-return (list FN VARs)) )
```

This assumes that the contents contain all used things. The semantics of CONTAINS will have to distinguish between shallow and deep containment.

PL:

```
[ OBJx (type obj 'fn)
       [ Xx (type x 'var) (uses obj x)
            [ (sets obj x) [ (follows set use) ]] ]]

==>  [ FNx [ VARx (uses fn var)
                 [ (sets fn var) [ (follows set use) ]] ]]
==>  (uses fn Sfn) [ (sets fn Sfn) [ (follows set use) ]]
==>  (uses fn vare) [ (sets fn vare) [ use-->set ]]
```

NOTES:

FOLLOWS is a functional.

Assignment (sets) is itself a type of function, so the only FN that both USES and SETS is assignment.

Assignment has a degenerate use of FOLLOWS.

If FN is a procedure, the meanings of USE and SET need clarification, like:

```
    All OBJ .· (type OBJ 'procedure) and
      Exist X Y . (type X 'var) and (type Y 'fn) and
                  (contains OBJ Y) and (contains Y X) and
                  (uses Y X) and
        if Exist Z . (type Z 'assign) and (contains Z X)
                                      and (sets Z X)
            then (follows Z Y)
```

Rewritten:

```
    All PROC Exist VAR FN .
      (contains PROC FN) and (contains FN VAR)
                         and (uses FN VAR)
      and if Exist ASSIGN . (contains ASSIGN VAR)
                            and (sets ASSIGN VAR)
          then (follows ASSIGN FN)
```

NOTES ON QUERIES:

1. The construction of a logical specification (and code) from the request takes these steps (reversing the development in the example):

   Request:

   ```
   [fn [lpe]]
   ```

   To relations:

   ```
   (contains fn lpe)
   ```

   To type identifiers:

   ```
   (type obj 'fn) (type Sobj 'lp) (contains obj Sobj)
   ```

   To conjunct/disjunct:

   ```
   (type obj 'fn) & (type Sobj 'lp) & (contains obj Sobj)
   ```

   To elemental code:

   ```
   (if (and (type obj 'fn)
            (type Sobj 'lp)
            (contains obj Sobj) )
       (list (obj Sobj)) )
   ```

   To quantified code:

   ```
   (LOOP OVER ALL OBJ
      (LOOP OVER ALL SOBJ IN OBJ
           (if (and (type obj 'fn)
                    (type Sobj 'lp)
                    (contains obj Sobj) )
                (COLLECT SOBJs) )
           (COLLECT-AND-RETURN
                (list (obj SOBJs)) ) )
   ```

2. Since CONTAINS is an elementary relation, it is confusing to simplify away the containment, by using the rule of double negation, [[a]] ==> a.

   ```
   [[ fn [lp] ]]  =/=>  fn [lp]
   ```

   The left-hand-side reads FN AND NOT LP, which is not what we want to express.

   The representation of NOT CONTAINS can be an elaboration of CONTAINS. This could be the double bracket, or it could be a highlight of the entire containment structure.

3. The PL form here is the same as in #11, but the item returned is different. Non-existent objects cannot be returned, so no confusion can result. However the information about what to return must still be displayed. In general, the object that organizes the returned values (the first one in the result list) should be highlighted in the display.

   In the abstract form:

   ```
   [ a [ b [c] ] ]
   ```

   one item will be special. Let the highlight be ASTERISKS here:

   ```
   [ a [ *b* [c] ] ]
   ```

   which means that All B are to be returned. Requests are Universal rather than Existential, unless explicitly labeled otherwise. (The request "Find a function with a loop" isn't highly motivated.)

   So the default quantification of the asterisked token is ALL.

   Although CONTAINS can be treated symmetrically for its two arguments, NOT-CONTAINS is not symmetric. Specifically,

   A contains B
   B contained-in A
   A not-contains B

   are all determined by looking inside of A.

   B not-contained-in A

   is determined by looking both inside (to assure B is not there) and outside (to find the B) of A.

4. ELLIPSIS and AMBIGUITY: Natural English is often not sufficiently specific to parse into a formal language. The Instructability strategy should be used in these cases.

   Psychologically, all requested objects should assume existence. #14 could be interpreted "if the FN contains an LP then the LP contains an IF." Under this interpretation, FNs that do not contain LPs would be returned, as well as FNs with LPs with IFs. The former group should be requested separately (its hard to imagine wanting both groups for the same reason).

B. Use of the Mock Engine

B.1 Starting the System

To begin using the Mock Engine (henceforth known as ME), the user should type a sequence of commands to the Symbolics LISP Machine:

- (load "A:>susan>mockeng>mocksys.lisp"): This command will load the defsystem for the ME.

- (make-system 'mockeng :noconfirm): This command will load all of the binary files that compose the ME. Ignore the notice that there is an unknown special instance variable, zwei:*last-file-name-typed*.

- (pkg-goto 'mockeng): Change the package to mockeng.

- (make-mock): This command will turn the mouse from its usual arrow form to a top-left corner of a rectangle form. Position the mouse blinker in the top left-hand corner of the screen and click left. Next, move the mouse until the elastic window fills the entire screen and click left again. This will be the window used for the ME interactions. Control will be returned to the user in the lisp listener.

- (top-level): The ME frame will be exposed and the user can begin making interesting drawings.

B.2 General Screen Layout

The ME screen is divided into four windows, as can be seen in Figure B-1. The large window found in the top-left position is the graphic display window that will contain the graphic picture being developed. The window underneath that is the interaction window. The user will be prompted in this window when either an action or some input is needed. There are two menu windows. The one on the top right of the screen is the object menu. It contains the commands that operate on the individual objects, both in their creation and later manipulation. The bottom menu contains commands that operate on the screen as a whole, including clearing the screen or saving its contents.

Following is a more complete definition of the menu choices and their consequences:

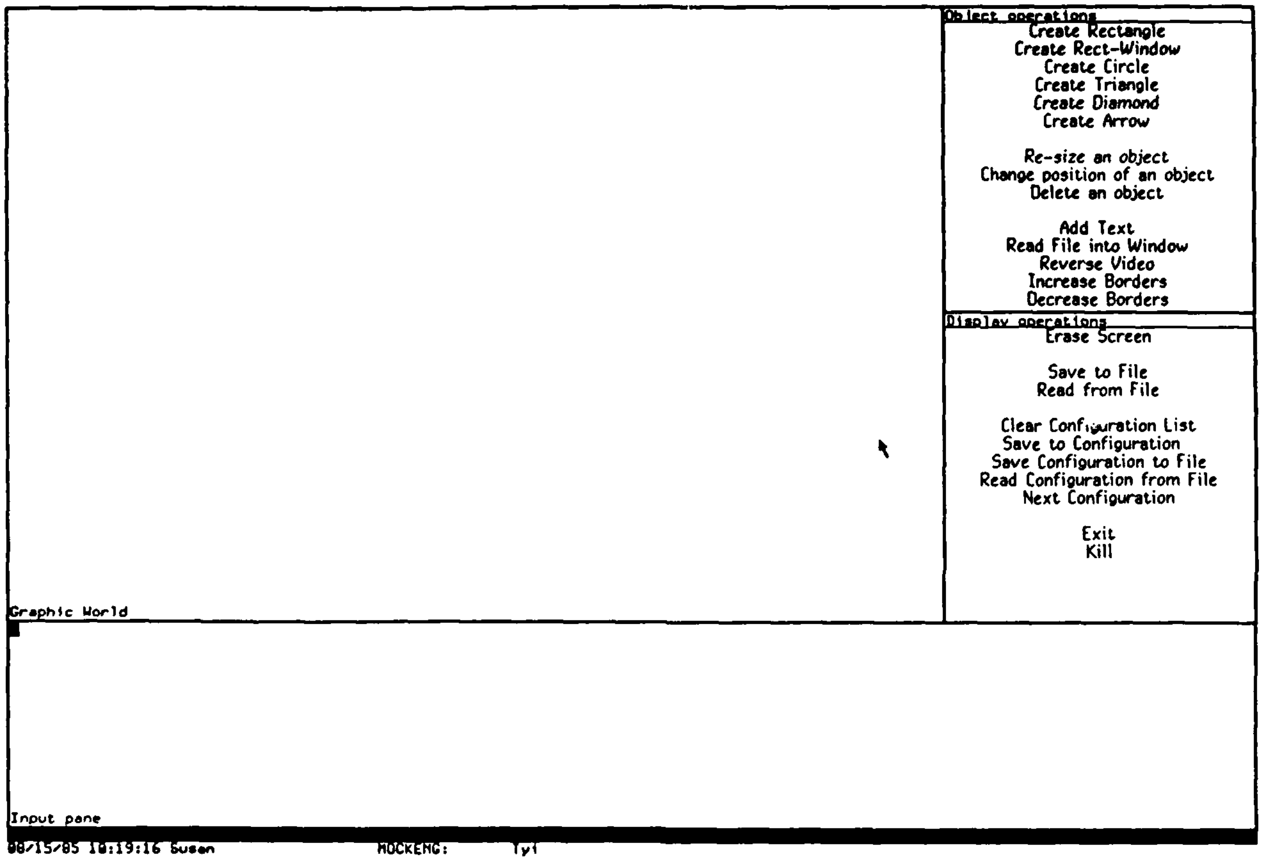


Figure B-1: Screen Design

B.3 Object Menu

Figure B-2 is an example of the Mock Engine in use. The object menu consists of the following commands and usages. (You will see a short explanation of the action of the menu choice in the mouse documentation line.) In any of the create-object options, a small object will be created that the user will drag around on the screen and initially place by a left-click. The first list of commands contains those that create objects.

- Create Rectangle: creates a new rectangle (note: this is just a basic rectangle, not an editable one)

- Create Rect-Window: creates a "rectangle-window". A left-click on this item creates a window whose superior is the entire graphics window. A right-click is needed to create a window inside an already existing rectangular-window. In this case, the user will first need to select the superior window before setting the position of the new window.

- Create Circle: creates a circle

- Create Triangle: creates a triangle

- Create Diamond: creates a diamond

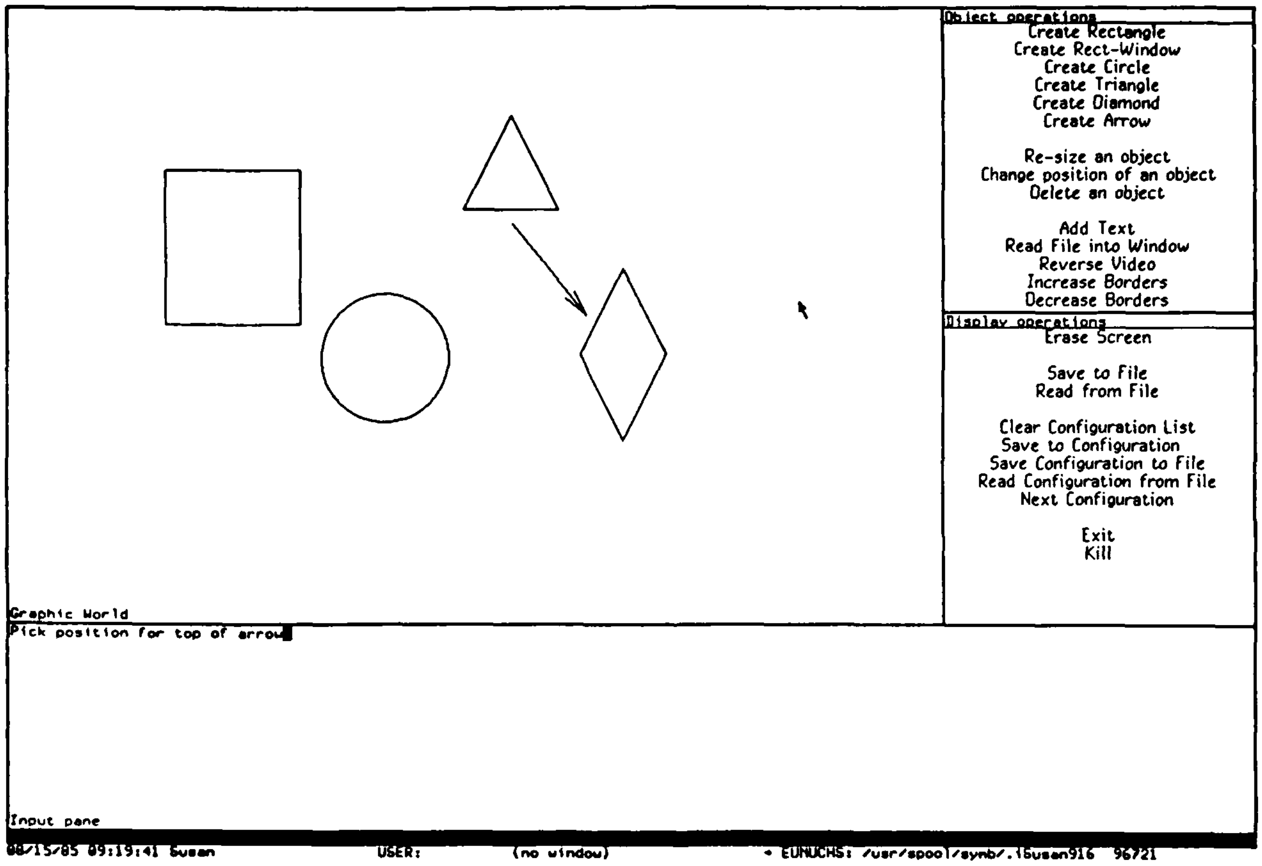
- Create Arrow: creates an arrow. The user will first be prompted for the head object to use for the arrow and then for the tail object for the arrow. The arrow will be drawn from the midpoint of an edge of the head object to the midpoint of an edge of the tail object.

The next set of commands are those that operate on any of the object types. The selection of an object is accomplished by moving the mouse blinker over the desired object. The system will indicate that the mouse blinker is over an existing object by drawing a rectangular outline around it.

- RE-SIZE AN OBJECT: changes the size of an existing object. The user will first be asked to select the object to be modified and will then be given a small object-like thing to expand or contract until the desired size is obtained.

- CHANGE POSITION OF AN OBJECT: move an object to a new position. The user will select the object to be moved and then be able to drag it around to its new position.

- DELETE AN OBJECT: delete an object. The user will select the object that is about to die and then it will cease to exist.



```
Pick position for top of arrow
```

```
08/15/85 09:19:41 Susan          USER:       (no window)          • EUNUCHS: /usr/spool/synb/.15usan916   96721
```

Figure B-2: Sample Mock Engine Usage

The commands that follow are those that operate only on the "rectangular-windows".

- ADD TEXT: add text to a window. The user will select a window and then be able to type text into that window using all of the favorite EMACS commands. Once satisfied with the window's contents, the user should hit the <END> key.

- READ FILE INTO WINDOW: read a file into a window. The user will first be prompted to select a window and then will be asked to input the name of a file to be read into that window.

- REVERSE VIDEO: reverse the video on a window. The user will select a window whose background color should reverse itself from white to gray or from gray to white.

- INCREASE BORDERS: increase the border width of a window. The user will select a window whose border width should be doubled.

- DECREASE BORDERS: decrease the border width of a window. The user will select a window whose border will return to the default width.

B.4 Screen Menu

The screen menu consists of those commands that are specific to operation on the entire screen. They are:

- ERASE SCREEN: clear the graphics window. This command both clears the graphics window and deletes all of the objects that it may have contained.

- SAVE TO FILE: save the current graphic display on a file. The user will be prompted for a file name on which to save the graphic objects that form the current picture.

- READ FROM FILE: read the graphic display from a file. The user will be prompted for a file name from which to read one graphic display.

The next group of commands operate on what is called the "configuration list." This list is an internal list that can contain different graphic configurations for saving and later display. The user will use this list to save a group of graphic displays. He will then save this grouping on a file and later be able to read that file back into the "configuration list" for a slide-presentation effect of flipping through a series of pre-defined displays.

- CLEAR CONFIGURATION LIST: clear the configuration list. Clear the configuration list of its contents.

- SAVE TO CONFIGURATION: save the current graphic display in the configuration list. The objects in the current screen display will be saved.

- SAVE CONFIGURATION TO FILE: save the current configuration list to a file. The user will be prompted for a file name on which to save a group of screen displays.

- READ CONFIGURATION FROM FILE: read the configuration list from a file. The user will supply a file name of pre-designed displays. The file will then be read into the configuration list.

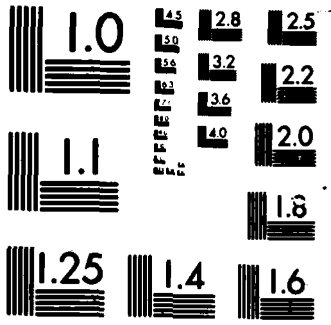
- NEXT CONFIGURATION: get the next configuration the configuration list. The next display that was stored on the configuration list will be displayed on the graphics window.

The last two commands are strictly general system commands.

- EXIT: exit the program. Returns control to the lisp system. The user can return to the lisp listener by typing a <select> <L>. Later, if he wants to return to the ME, he will type (top-level) and will be placed back into the state that he left.

- KILL: kill the window. Kill the ME and exit the program.

AD-A163 180   FORMALIZATION OF THE INTELLIGENT PROGRAM EDITOR(U)   2/2
ADVANCED DECISION SYSTEMS MOUNTAIN VIEW CA
S G ROSENBAUM ET AL. DEC 85 TR-1047-02 N00014-83-C-0444
UNCLASSIFIED   F/G 9/2   NL


1.0
4.5
5.0
5.6
6.3
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

C. Sample Demo of Text/Syntax Linkages

We have a primitive system running now with text and syntax representations cross-linked. In this section, we describe several pictures of that system in order to give the reader a flavor for how the system currently works.

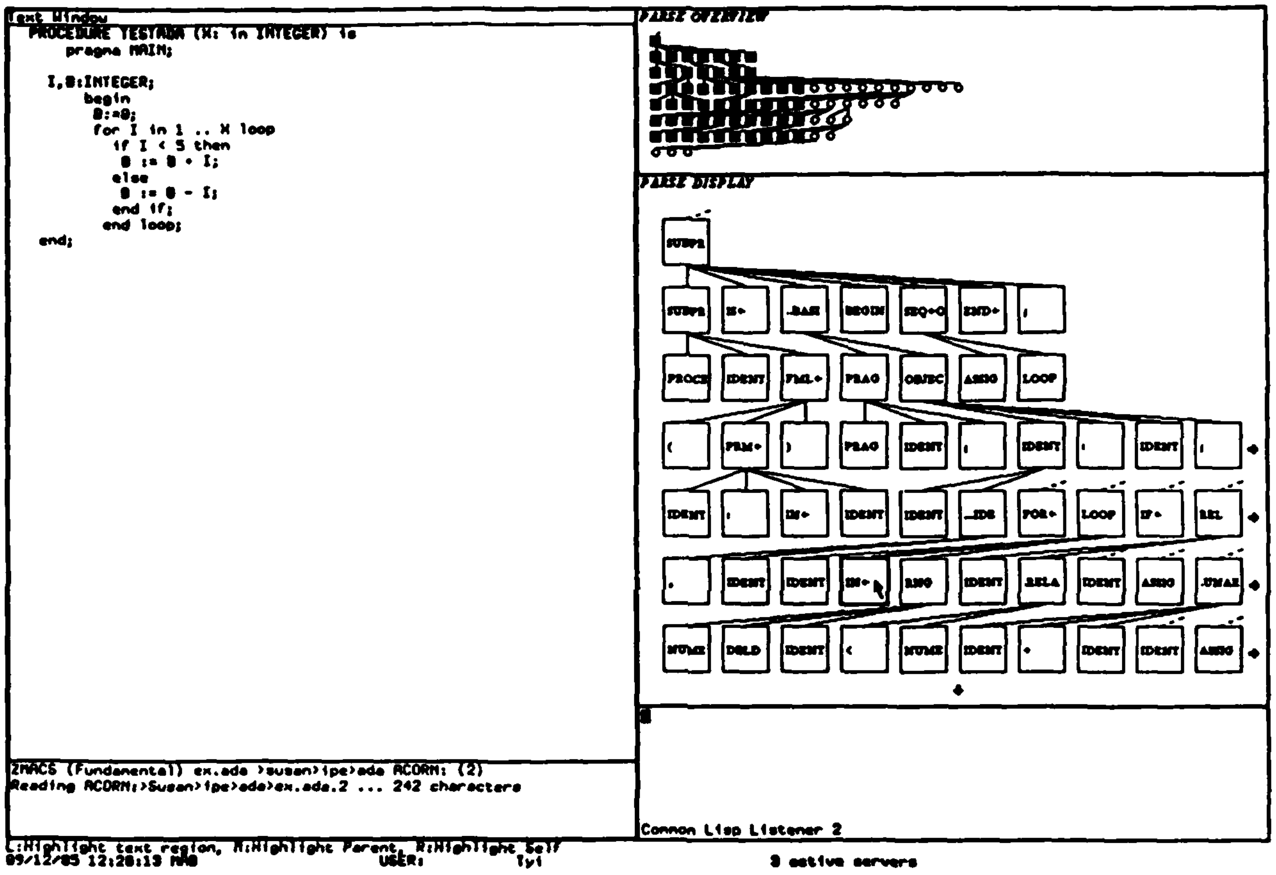
Figure C-1 is a picture of the Lisp Machine screen after starting the system. The left side of the screen is dedicated to the text representation of the program. For this portion we are using the ZMACS text editor with appropriate modification to interface with the rest of our system. The right side of the screen is divided into three sections. The middle one is the actual parse tree displayed as described earlier. The root node is always a single node at the top of this window. Each box is a syntax node and is labeled with the first few letters of the node type. The horizontal and vertical arrows that appear at the right and left edges of this screen are *scroll icons*--they indicate that there is more parse tree in that direction which is not shown because of lack of space. If a scroll icon is *moused*, the tree will scroll in that direction. Because the entire tree is not visible at once, above the parse display is another window which provides an overview of the parse tree. In the overview display, the entire tree is always visible but, due to size constraints, unreadable. The highlighted nodes (those which are blacked out) are the ones that are currently being displayed in the parse tree window below. The overview window gives the complete shape and structure of the parse tree, while the parse window is used to see and manipulate the individual nodes. The overview can be thought of as a road map to the parse display. Finally, the bottom window on the right side of the screen is a Common LISP window which allows us to develop and debug the system more effectively. This window would *not* appear in a production version of the IPE.

In addition to the keyboard, the Lisp Machine also has a *mouse* which is used by the user to command the Lisp Machine. The mouse is used to drive a cursor around the screen. In these screen images, the mouse cursor appears as a diagonal arrow, pointing up and left. In figure C-1, the mouse cursor is in the parse display window, in a node of type "IN<-" which is in the second to last row, fourth column counting from the left. Because the mouse is pointing to that node, the node has a thickened border. The mouse has three buttons, and in this system, when it is over a syntax node, the right button will highlight the node it is over, the middle button will highlight that nodes parent, and the left button will highlight the region of text in the text window which is pointed to by the selected syntax node.

The second picture (figure C-2) shows how this works. Both the node that the mouse is over and its parent have been highlighted. Sometimes the parent of a node is *not* currently on the screen. In that case, the line that the parent is on should be scrolled so that the parent is visible.

In figure C-3, we have returned the parent of the highlighted node to normal display state and have additionally highlighted the text in the text window which corresponds to that node, namely "in".

From within the text/editor window on the left side of the screen, we can, as mentioned above, select a region of text and see which syntax nodes refer to it. The operation which highlights a region of text that is referenced by a specific syntax node (in the right center window) also selects that region of text. Once that is done, all syntax nodes that refer to that piece of text can be highlighted at once. This is how figure C-4 was made. The highlighted syntax nodes are all those nodes that refer to the same piece of program text that is referenced by the node the mouse has selected. Since in this case it is a leaf node, this should be all the ancestors of the node back to the tree root. In figure C-4, there is apparently one missing in the fourth row. Figure C-5 shows the reason--the highlighted node in that line was off the screen. In this figure, the mouse has been moved to the scroll icon for that row, which was then scrolled. Notice the changes in both the parse display and the parse overview windows.

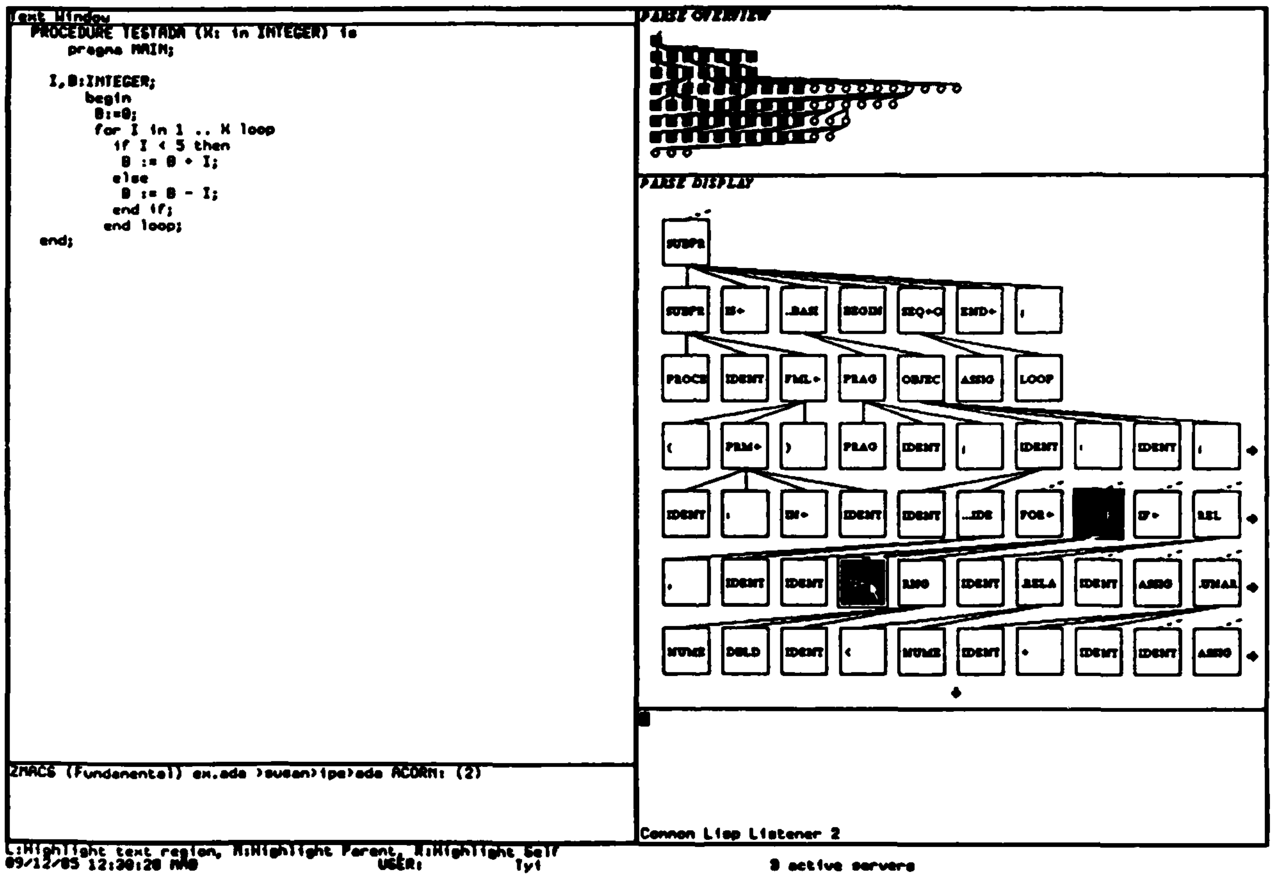


```
  PROCEDURE TESTADA (X: in INTEGER) is
      pragma MAIN;

    I,B:INTEGER;
        begin
          B:=0;
          for I in 1 .. X loop
            if I < 5 then
             B := B + I;
            else
             B := B - I;
            end if;
           end loop;
   end;
```

```
ZMACS (Fundamental) ex.ada >susan>ipe>ada ACORN: (2)
Reading ACORN:>Susan>ipe>ada>ex.ada.2 ... 242 characters
```

```
L:Highlight text region, M:Highlight Parent, R:Highlight Self
09/12/85 12:28:13 MAB              USER:        Tyi                    3 active servers
```

Figure C-1: Start-Up Screen



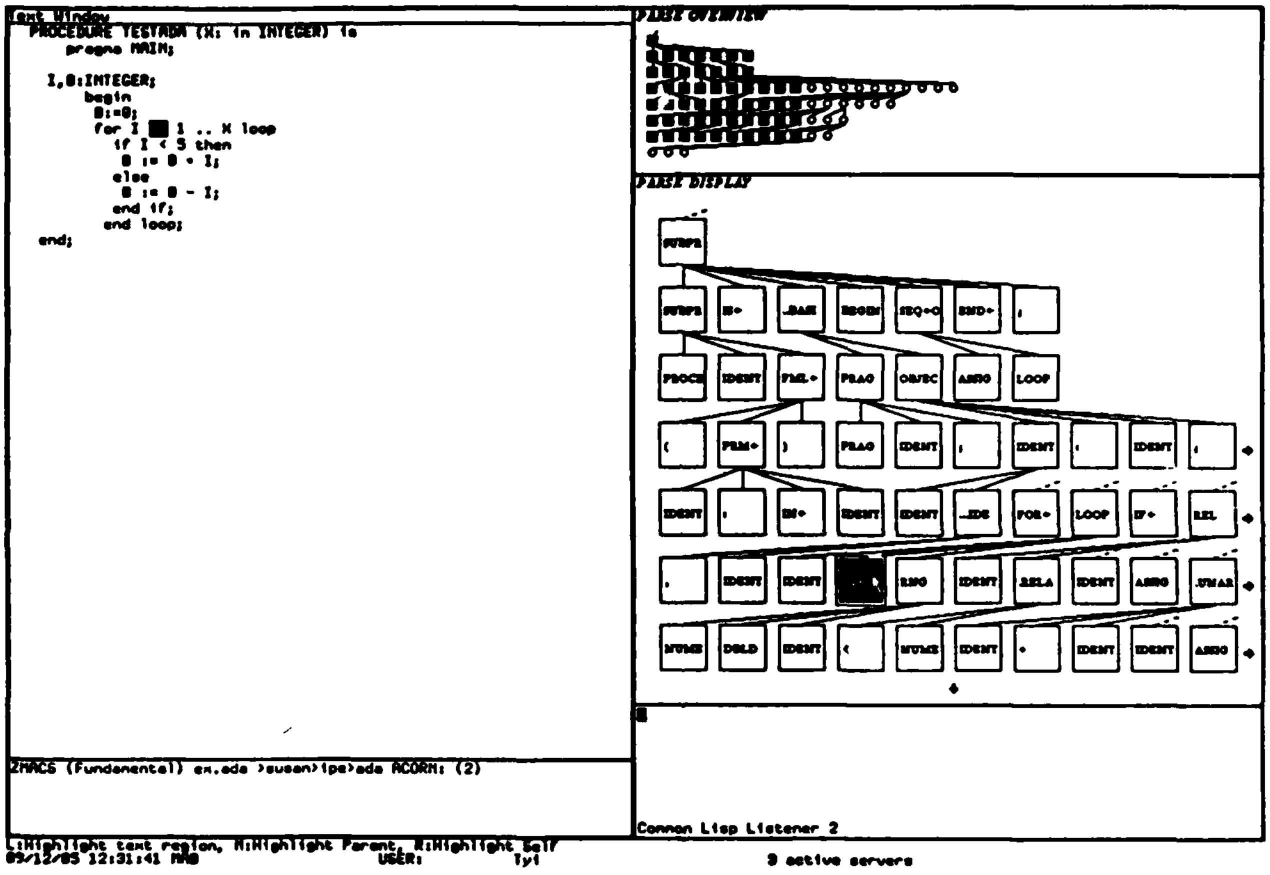
```
PROCEDURE TESTADA (X: in INTEGER) is
    pragma MAIN;

  I,B:INTEGER;
      begin
       B:=0;
       for I in 1 .. X loop
         if I < 5 then
          B := B + I;
         else
          B := B - I;
         end if;
        end loop;
  end;
```

```
ZMACS (Fundamental) ex.ada >susan>ipe>ada ACORN: (2)
```

```
Common Lisp Listener 2
```

```
L:Highlight text region, M:Highlight Parent, R:Highlight Self
09/12/85 12:30:28 MAG              USER:       Tyi                          3 active servers
```

Figure C-2: Parent Node

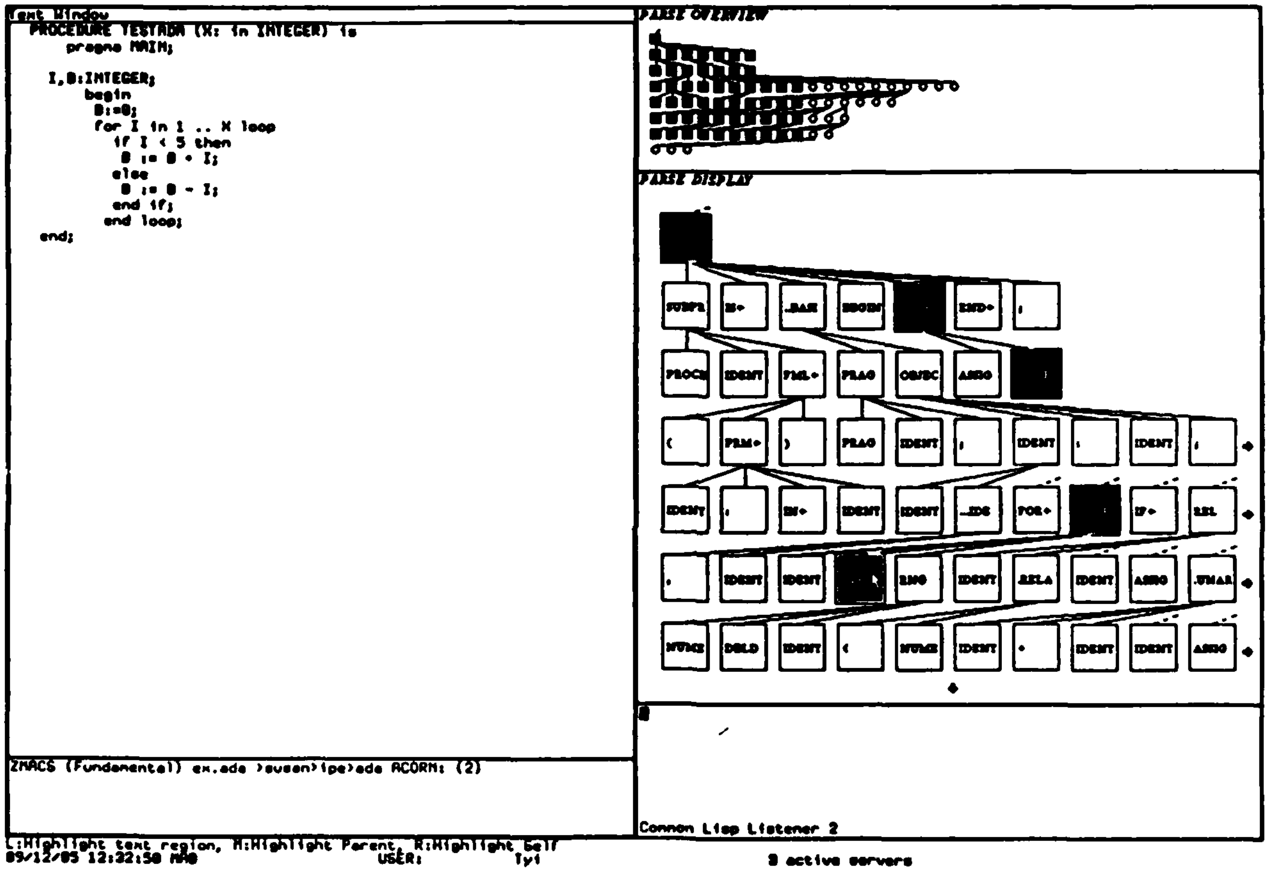


```
  PROCEDURE TESTADA (X: in INTEGER) is
      pragma MAIN;

    I,B:INTEGER;
        begin
         B:=0;
         for I █ 1 .. X loop
           if I < 5 then
            B := B + I;
           else
            B := B - I;
           end if;
          end loop;
   end;
```

```
ZMACS (Fundamental) ex.ada >susan>ipe>ada ACORN: (2)
```

```
L:Highlight text region, M:Highlight Parent, R:Highlight Self
09/12/85 12:31:41 MAB                  USER:            Tyi                           9 active servers
```

Figure C-3: Corresponding Text



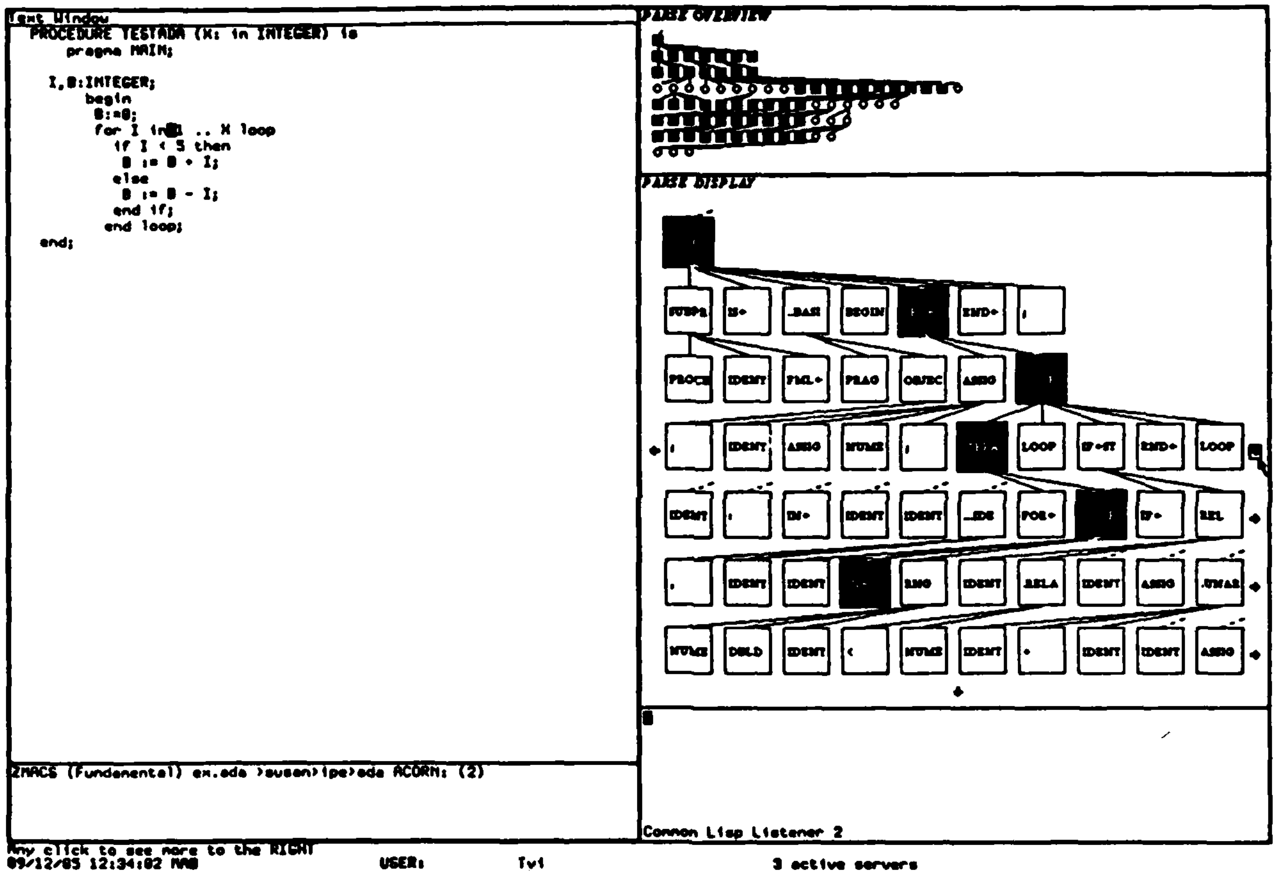
```
  PROCEDURE TESTADA (X: in INTEGER) is
      pragma MAIN;

    I,B:INTEGER;
        begin
         B:=0;
         for I in 1 .. X loop
           if I < 5 then
            B := B + I;
           else
            B := B - I;
           end if;
          end loop;
   end;
```

```
ZMACS (Fundamental) ex.ada >susan>ipe>ada ACORN: (2)
```

```
Common Lisp Listener 2
```

```
L:Highlight text region, M:Highlight Parent, R:Highlight Self
09/12/85 12:22:50 MAB                USER:          Tyi                     9 active servers
```

Figure C-4: Corresponding Syntax



```
  PROCEDURE TESTADA (X: in INTEGER) is
      pragma MAIN;

    I,B:INTEGER;
        begin
         B:=0;
         for I in 1 .. X loop
           if I < 5 then
            B := B + I;
           else
            B := B - I;
           end if;
          end loop;
   end;
```

```
ZMACS (Fundamental) ex.ada >susan>ipe>ada ACORN: (2)
```

```
Common Lisp Listener 2
```

```
Any click to see more to the RIGHT
09/12/85 12:34:02 NAB            USER:         Tv1                    3 active servers
```

Figure C-5: Scrolling the Syntax Tree

References

[Bricken 85] W. Bricken, S. Rosenbaum, M. Brzustowicz, J. Dean, & B. McCune.
Formalization of the Program Reference Language.
Technical Report TR-1066-01, AI&DS, October, 1985.

[Brzustowicz 84] M. Brzustowicz, J. Dean, B. McCune, & S. Rosenbaum.
Annual Report for the Intelligent Program Editor.
Technical Report TM-1047-1, AI&DS, November, 1984.

[Domeshek 84] E. Domeshek, J. Dean, S. Rosenbaum, & B. McCune.
Design of a Pictorial Program Reference Language.
Technical Report TR-1014-4, AI&DS, August, 1984.

[Shapiro 83] D. Shapiro & B. McCune.
Searching a Knowledge Base of Programs and Documentation.
Technical Report TM-1014-2, AI&DS, January, 1983.

[Stallman 81] R. Stallman.
EMACS Manual for ITS Users.
Technical Report AI Memo 554, Massachusetts Institute of Technology Artificial Intelligence Laboratory, October, 1981.